Debreceni Egyetem

Informatikai Kar

A Spring Web Flow és a JSF 2.0. összehasonlítása

Témavezető:

Dr. Adamkó Attila

egyetemi tanársegéd

Készítette:

Kovács Tibor Krisztián

II. programtervező informatikus (MSc)

Debrecen

2010

Tartalomjegyzék

Előszó és köszönetnyilvánítás

A dolgozatom célja középhaladó szinten beavatni az olvasót a Spring és a JSF keretrendszerek működésébe, de főleg az azok nyújtotta lehetőségekbe. Már több mint egy éve gyakornokként dolgozom az Orgware Kft.-nek ahol főleg a Spring keretrendszer megismerésével foglalkoztam, ezért előre bocsájtom, hogy a dolgozat inkább Spring centrikus lesz. Továbbá mivel mindkét keretrendszer jelentősebb méretű, főleg a Spring, ami egyenesen óriási, nagy valószínűséggel több dolog is ki fog maradni a leírásokból, amit remélhetőleg senki nem fog felróni nekem. Habár a dolgozat címe összehasonlítás, biztosan nem fogok állást foglalni egyik technológia mellett sem, inkább csak szembe állítom a két keretrendszer adta különböző eszközöket.

Köszönetet szeretnék mondani témavezetőmnek, Dr. Adamkó Attilának, aki hasznos tanácsokkal és észrevételekkel segítette a szakdolgozatom megírását. Továbbá a külsős cégemnek az Orgware Kft.-nek is, akiknél betöltött gyakornoki állásom nélkül neki se tudtam volna fogni ennek a dolgozatnak.

1. Bevezetés

Mint az előszóban is írtam, a dolgozattal inkább a középhaladó szintet szeretném elérni, ebből kifolyólag szeretném elhagyni azokat a dolgokat, amiket már a legtöbb szakdolgozat bevezetőiben már sokan leírtak. Ilyenek lennének az internet fejlődése, a különböző technológiák kialakulása, felemelkedése, majd bukása is. Manapság minden egyetemista és cég tisztában van vele, hogy a jelenlegi jelentős technológiai irányzatok a vállalati rendszerek és azok integrációja körül forog.

A Spring is az előbbi bekezdésben említett vállalati rendszerek létrehozására lett kifejlesztve, azon belül is főleg a kis és középvállalatoknak nyújt alternatív keretrendszert, főleg az EJB-kkel szemben. Ez azt jelenti, hogy a Springnek korlátozott eszközkészlete van a nagy elosztott rendszerek fejlesztésére, helyette inkább a kisebb rendszerek integrációjára, régi maradványkódok újrahasznosítására, és e kódok együttes használatára fekteti a hangsúlyt a legújabb technológiákkal. A lehető legtöbb, könnyen használható eszközt próbálja nyújtani a különböző technológiák támogatására, azok együttműködésének a biztosítására, miközben a lehető legkevesebb függőséget kell kialakítani a keretrendszer API-jaira építve.

A Spring Web Flow a Spring egy része, ami a webes alkalmazások navigációjának egyszerűsítésére hoztak létre. Kevésbé rejtett tény, hogy a JavaServer Faces technológia „ellenfelének” szánták, annak hiányosságai miatt. Azóta kijött a JavaServer Faces legújabb verziója, a 2.0, ami rengeteg újítást tartalmaz. Ezeket az újításokat majd meg is fogom tüzetesebben vizsgálni a szakdolgozat második felében. A Spring Web Flow egyébként csak a navigációra tartalmaz logikát, amit nem csak a webalkalmazásokban lehet használni, hanem akár az asztali alkalmazásokban is. Ebből kifolyólag, az SWF-et úgy tervezték, hogy tetszőleges megjelenítési technológia mellett használható maradjon. Épp ezért, a szakdolgozatom mellé leadott programban is az SWF-et a JSF 1.2-vel fogom használni, de a JSF 1.2 csak a megjelenítésért lesz felelős. Azt hogy a két technológia hogyan működik együtt, majd az SWF-ről szóló fejezetben leírom. Az összehasonlítás esetében is legtöbbször a JSF 1.2 úgy lesz jelen, mintha mindkét technológia arra építene. Ez azért tehetem meg, mert a megfelelő beállításokkal a két technológia között majdhogynem tetszőleges mértékű átjárás lehetséges, továbbá a JSF 2.0 természetesen minden eszközt örököl az 1.2-ből.

A szakdolgozatomban nem lesz szó eszközök installálásáról, és az alapvető könyvtárstruktúra kialakításáról sem (kivéve ahol szükséges), továbbá a webalkalmazások építéséről (bulid), és azok webalkalmazás szerverre történő publikálásának (deploy) mikéntjéről sem lesz szó, ha valaki nincs tisztában ezekkel a tevékenységekkel, az interneten, vagy az irodalomjegyzékben hivatkozott könyvekben könnyen és gyorsan informálódhat róla.

2. Java Server Faces

A Java Server Faces-ről először 2002-ben lehetett hallani a JavaOne konferencián. Annak ígéretében jött létre, hogy a webalkalmazások programozásának kellemetlenebb, servlet és JSP programozási részét egy barátságos felülettel váltsák fel, hogy a programozók ezen túl a kérések és az oldal navigációk megoldása helyett inkább szöveges mezőkben és menükben gondolkodjanak. A szakma fel is pezsdült ezek hallatán és 2004-ben meg is jelent a JSF 1.0, majd a hibajavítások, kódtisztítások és néhány új kényelmi funkció bevezetésével 2006-ban kijött a JSF 1.2.

Természetesen amennyire jól hangzottak az ígéretek annyira voltak nagyok a csalódások is. Sajnálatos módon a JSF 1-es verziót mintha elefántcsonttoronyban fejlesztették volna, rengeteg hiányossággal és nehézséggel küzdött. Viszont az ötlet maga kiváló volt, így a szakma gyorsan a való világban is jól alkalmazható projektekkel támogatta meg a specifikációt. Ezek főleg a harmadik fél által fejlesztett komponenskönyvtárak voltak, valamilyen fontosabb technológia támogatásával. Ilyen könyvtár például az IceFaces, vagy a MyFaces Trinidad, mindkettő komoly AJAX támogatással.

A JSF azon az ötleten alapszik, hogy a felhasználói interfészt komponensek HTML űrlapokra húzásával lehessen fejleszteni, majd a komponensekhez Java objektumokat rendelni, így elkülönítve a markup és java kódokat (akár lehet úgy is gondolni rá, mintha ez lenne a webalkalmazások Swing-je). A JSF további erőssége a jól bővíthető komponens modell, ami felhasználásával sok ingyenes, és nem ingyenes komponens könyvtárt hoznak létre. És végül a JSF támogatja az üzleti logika és megjelenítés szétválasztását (azaz az Modell-View-Controller tervezési minta könnyen megvalósítható vele.), az oldalak közti navigációt, és a külső szolgáltatásokhoz való kapcsolódást is.

A Java Server Faces-nek három fő részét szokták megkülönbözetni:

- Előre gyártott UI komponensek egy halmaza.
- Eseményvezérelt programozási modell.
- Komponensmodell a kívülálló harmadik csoport fejlesztői számára, további komponenseket készítéséhez

Ezzel szemben a JSF alkalmazásnak négy fő része szokott lenni:

- Konfigurációs fájlok a szerver konfigurálásához.
- Konfigurációs fájlok a JSF konfigurálásához.
- Beanek a felhasználói adatok kezeléséhez.
- JSP vagy JSF oldalak a felhasználói interfészhez.

2.1. Konfiguráció

A JSF konfigurálásához először a servlet konfigurációt érdemes elkészíteni, azaz be kell állítani a Faces Servlet-et és a hozzá tartozó servlet mappingot a web.xml fájlban (2.1. kódrészlet). A servlet mapping-ban megadott megfelelő prefix vagy suffix felelős azért, hogy a beérkező URL-ek a megfelelő servleteket aktiválják (a servletek a beérkező kérések feldolgozásáért felelnek). A JSF oldalak kéréseit ugyanis egy speciális JSF servletnek kell feldolgoznia, nevezetesen az előbb említett Faces Servlet-nek. Ezt a servlet osztályt minden JSF implementációs kódnak tartalmaznia kell.

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

2.1. kódrészlet

Ezen beállítások mellett a Faces Servlet automatikusan .jsp végződésű oldalakat fog keresni a szerveren telepített (deploy) könyvtárakban. Ahhoz, hogy ezt a viselkedést megváltoztassuk, egy környezeti változó beállítása szükséges ugyanebben a fájlban (2.2. kódrészlet).

```
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.jspx</param-value>
</context-param>
```

2.2. kódrészlet

A JSF beállításait alapértelmezett esetben a WEB-INF könyvtárban található faces-config.xml fájl tartalmazza (átállítható a web.xml fájlban a 2.3. kódrészlet szerint). Ebben a fájlban találhatóak az alkalmazásra (ezen belül lehet megadni message bundle-t, saját hiba üzeneteket biztosító osztályt, alapértelmezett render kit-et, valamint az Expression Language (EL) változók feloldását végző osztályokat is. Az EL segítségével a beanekben tárolt adatokat érhetjük el egyszerűen a nézeteken), beanekre, navigációra, validációra, konverterekre, komponensekre, render kitekre és életciklusra vonatkozó beállítások.

```
<context-param>
  <param-name>javax.faces.CONFIG_FILES</param-name>
  <param-value>>WEB-INF/something.xml,WEB-INF/another.xml</param-value>
</context-param>
```

2.3. kódrészlet

A JSF a faces-config.xml konfigurációs fájl szintaktikájának megadására az 1.2-es verziótól séma deklarációt használ. (2.4. kódrészlet)

```
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">

</faces-config>
```

2.4. kódrészlet

2.2. Beanek

A Java beaneknek nevez minden olyan osztályt, amely tulajdonságokat és eseményeket tesz elérhetővé külső környezet számára (Pontos definíció: újrahasználható szoftverkomponens, ami manipulálható fejlesztőeszközökben http://java.sun.com/products/javabeans/). A tulajdonság egy adott típus megnevezett értéke, amit írni, vagy olvasni lehet. Egyszerűbben fogalmazva az osztály olyan adattagokat tartalmaz, amihez getter és/vagy setter metódus tartozik, valamint magában foglal egy paraméter nélküli konstruktort is.

A JSF alkalmazásokban az összes olyan adat számára beaneket használnak, amik megjelenhetnek a felhasználói felületen. A beanek csatornák a felhasználói interfészek és az alkalmazás logikája között. Pontosabban leírva a beanek a következő főbb célokat szolgálják: felhasználói felületek komponensei, összekapcsolják a webes formokat és azok viselkedését, az oldalon megjelenő üzleti logika objektumaiként szolgálnak és végül szolgáltatásokként is felhasználhatóak. Speciálisan a JSF BackingBean-nek nevezi az olyan beaneket, melyek az oldal komponenseihez tartozó objektumokat adattagokként tartalmazzák. Fontos, hogy ha használunk egyáltalán BackingBeaneket, akkor se keverjük a komponensek adatait az üzleti logika modellobjektumaival.

Bean definiálása a faces-config.xml fájlban történik (2.5. kódrészlet). A bean, definiálása után, a jsp/jsf oldalakon hivatkozhatóvá válik value expression-ök segítségével. A beanek osztályában a `@PostConstruct` és `@PreDestroy` annotációkkal a beanek készítése utáni és a törlése előtti kódokat adhatjuk meg.

```
<managed-bean>
  <managed-bean-name>backingBeanName</managed-bean-name>
  <managed-bean-class>backing.bean.package.BeanClass</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

2.5. kódrészlet

A beanek definiálásakor megadhatunk függőségeket is. Ezeket a `<managed-property>` elemben írhatjuk le. A beállítani kívánt tulajdonság nevének megadása az előbbi elemen belül a `<property-name>`, érétkének beállítása a `<value>` vagy `<null-value>` segítségével történhet. A tulajdonságok beállításainak használatakor az összes megadott tulajdonság beállító metódusa meghívódik a megfelelő értékekkel rögtön a bean

inicializálása után (ami mindig a paraméter nélküli konstruktorral történik). Értéknek meg lehet adni Map vagy List típust is.

A webprogramozók kényelmére a servlet konténer különböző hatáskörökkel rendelkező táblázatot tart fent (scope), amik név és érték párokat kezelnek. Ezek a táblázatok tipikusan olyan beaneket és objektumokat tartalmaznak, amikre a webalkalmazás több részén is szükség van. A JSF 1.2-ben használható scopeok a következők:

- `request` Scope: Ez rendelkezik a legkisebb hatáskörrel, az itt elhelyezett objektumok csak a lekérdezés ideje alatt élnek. A `request` scopeban elhelyezett beanből minden lekérdezéskor új példány jön létre, és ez a példány törlődik a válasz elküldésekor.
- `session` scope: A következő, az előzőnél tágabb scope. A session az egyazon kliens által történő sorozatos csatlakozást jelenti. A servlet konténer minden egyes felhasználót nyomon követ a sessionazonosítók segítségével. Ezek az azonosítók vagy sütiken keresztül, vagy a kliensnél letiltott süti küldés esetén az URL-en keresztül kerülnek átadásra. Pontosabban megfogalmazva a `session` scope attól a pillanattól fogva tárolja az objektumokat, amikor egy kliens csatlakozik, addig a pillanatig, amíg a `session` meg nem szűnik. A `session` megszűnik, ha valami meghívja a HttpSession objektumon az `invalidate()` metódust, vagy a `session` az előre beállított futási időt túl nem lépi.
- `application` scope: A legtágabb scope. A webalkalmazás teljes időtartama alatt tárolja az objektumokat, és ezen a hatáskörön az összes `request` és `session` objektum osztozik. Az `application` scope-ban definiált bean a webalkalmazás bármely példányának első lekérésekor születik meg, és a webalkalmazás webszerverről való eltávolításakor szűnik meg.

2.3. Navigáció

A faces-config.xml segítségével konfigurálhatjuk a webalkalmazásunk navigációját. Itt állíthatjuk be, melyik oldalról melyik oldalra kerüljön át a felhasználó, az általa végrehajtott akciók és az üzleti logika által visszaadott értékek segítségével. Amikor a felhasználó megnyom egy gombot, az oldalon történt változásokat a böngésző elküldi a szervernek. Ezután a webalkalmazás analizálja a kapott adatokat, és dönt arról, melyik JSF oldal kerül megjelenítésre következőleg. A következő JSF oldal kiválasztását a NavigatonHandler interfész implementációja vezérli (navigáció-kezelő).

Statikus navigáció esetén a felhasználói felületen lévő gomb `action` attribútuma egy sima String. Ha a felhasználó a „`buttonAction`” akciót tartalmazó gombot nyomja meg a sourcePage.jspx oldalon, akkor a szerver válasza a nextPage.jspx oldalra való navigálás lesz. A `from-view-id` elem nélkül az összes oldalra érvényes lesz a navigációs szabály, de használatakor itt megadhatunk mintát is, szűkítve az oldalak számát.

```
<navigation-rule>
  <from-view-id>/sourcePage.jspx</from-view-id>
  <navigation-case>
    <from-outcome>buttonAction</from-outcome>
    <to-view-id>/nextPage.jspx</to-view-id>
  </navigation-case>
</navigation-rule>
```

2.6. kódrészlet

Amennyiben a gomb `action` attribútumában egy method expression (EL kifejezés típus) áll, dinamikus vezérlésről beszélünk. Ekkor method expressionben hivatkozott bean adott metódusa fut le, majd a metódus visszatérési értékét megkapja a navigáció-kezelő és ez alapján dől el a következő oldal. A hivatkozott metódusnak paraméter nélkülinek kell lennie, és tetszőleges visszatérési típussal rendelkezhet, mert a visszatérési érték konvertálódik a `toString()` segítségével.

Haladóbb technika a `<request>` elem megadása a `<navigation-rule>` elemben. Ekkor a navigációs szabályok által meghatározott következő oldalt nem küldi el a szerver a kliensnek, hanem csak egy redirect üzenetet a következő oldal címével, amit majd a kliens http GET metódus segítségével érhet el.

2.4. JSF oldalak

Minden böngészőben megjelenítendő oldalhoz szükséges egy külön JSF oldalra. Tipikusan ezeknek az oldalaknak .jsp kiterjesztése van, de ez változtatható, mint azt a 2.2. kódrészlet mutatja. A másik elterjedt kiterjesztés a .jspx, ekkor az oldalt xml segítségével adjuk meg, az xml szintaktikai és egyéb előnyeivel. A .jsp oldal tipikusan a tag könyvtárak megadásával kezdődik (.jsp odlal esetén 2.7., .jspx esetén 2.8. kódrészlet).

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
```

2.7. kódrészlet

```
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:f="http://java.sun.com/jsf/core">
  <jsp:output omit-xml-declaration="true" doctype-root-element="HTML"
              doctype-system="http://www.w3.org/TR/html4/loose.dtd"
              doctype-public="-//W3C//DTD HTML 4.01 Transitional//EN"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8"/>

</jsp:root>
```

2.8. kódrészlet

A JSF két fajta komponenshalmazt definiál. A HTML tag könyvtárakat (vagy névtereket), amik HTML specifikus kódot generálnak, és a core tag-eket, amiket viszont függetlenül a megjelenítő technikától mindig meg kell adni. Amennyiben más komponens könyvtárat használunk, mint a specifikációban megadott, akkor a HTML tag könyvtárat elhagyhatjuk, de további taglib definíciókat is hozzáadhatunk az eddig meglévőkhöz.

A jsp oldalakon nyugodtan használhatunk HTML és JSP kódokat, de csak a JSF elemeket használva is létrehozhatjuk az oldalt. A HTML/JSP kódoktól való főbb különbségek a következőek:

- Az összes JSF elemnek az `f:view` elemen belül kell lennie.
- A HTML `form` tag helyett használjuk a `h:form` elemet.
- Az `input` HTML elemek helyett használjuk például a `h:commandButton` a `h:inputText` és a `h:inputSecret` elemeket.

Nem sorolom fel az összes elemet és attribútumaikat, helyette bemutatok két általánosat. Az elemek legtöbb tulajdonsága hozzáköthető a beanek tulajdonságaihoz a value expression-ök segítségével.

```
<h:inputText value="#{beanName.property}"/>
```

2.9. kódrészlet

A 2.9. kódrészleten látható `inputText` tag a `beanName` néven deklarált bean `property` tulajdonságához van kötve. Mikor az oldal renderelésre kerül, meghívódik a `getProperty()` metódus a tulajdonság értékének lekérdezésére. Mikor a kliens visszaküldi az általa beírt adatokat, a `setProperty()` metódus fog meghívódni, hogy az új adatok mentésre kerüljenek a bean-ben.

```
<h:commandButton value="Button" action="action"/>
```

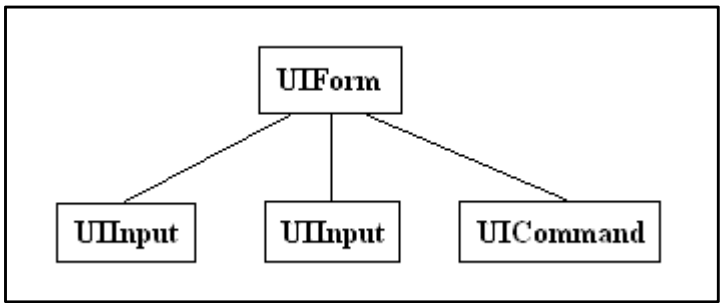
2.10. kódrészlet

A 2.10. kódrészletben a `commandButton` tag `action` attribútumában látható érték segít eldönteni a navigációt. Mikor a kliens rányom a gombra, ez az attribútum belekerül a lekérésbe, majd a JSF felhasználja a navigálásra. Az `action` attribútum is lehet value expression, ekkor a hozzá kötött metódus visszatérési értéke kerül felhasználásra a navigáció során, mint az előző fejezetben már említettem, ekkor dinamikus navigáció történik.

2.5. Komponens fa

Már minimálisan felvázoltam az alapokat, így most vizsgáljuk meg a JSF működését is. Induljunk onnan, mikor a felhasználó beírja az alkalmazásunk címét a böngészőjébe, mondjuk a http://localhost:8080/sample/faces/firstpage.jsp URL-t. A szerverhez beérkezik a kérés, és a servlet mapping alapján eldől, hogy a FacesServlet kezeli a kérést. Mivel ez az első ilyen kérés, a JSF servlet beolvassa a firstpage.jsp fájlt. Az oldal tartalmazza az `f:view` elemet, és például további két `h:inputText` és egy `h:commandButton` elemet. Minden egyes elemnek van egy azt kezelő osztálya, így mikor egy tag beolvasásra kerül, az azt kezelő osztály végrehajtásra kerül, és a tagkezelő osztályok egymással közreműködve felépítik a komponensfát (2.11. ábra). A komponensfa olyan adatstruktúra, amely a JSF oldalon található minden egyes felhasználói felület-elem számára tartalmaz egy objektumot. Az összes JSF

komponens kap egy egyedi azonosítót, hogy azok megkülönböztethetőek legyenek. Amennyiben nem adtunk meg azonosítót, a rendszer automatikusan kioszt egyet.



2.11. ábra. JSF komponensfa

A komponens fa felépítése után az oldal renderelődik, az összes nem JSF szöveg változás nélkül átíródik a kliensnek küldendő HTML üzenetbe, míg a JSF elemek HTML kódra konvertálódnak (például az elem azonosítója a HTML elemek `name` attribútumába kerül). Ezt a folyamatot kódolásnak nevezik. A kódolást vagy a komponens maga végzi, vagy a renderelő osztálya. A kódolás végén a JSF oldalból készült HTML oldal elküldésre kerül, és a kliens böngészője megjeleníti azt.

A kliens miután kitöltötte azokat a mezőket, amiket szeretett volna, és egy gombra kattint, a gombot tartalmazó űrlapot és a beírt adatokat a böngésző visszaküldi a szervernek, mint http POST kérést. A visszaküldött adatok azonosító és érték párok, melyeket a konténer normál működés alapján elment egy hash táblába. A JSF keretrendszer lehetőséget biztosít a komponenseknek, hogy szabadon böngésszék ezt a táblázatot, és dönthetnek arról, hogy használják fel a visszaküldött űrlap adatait. Ezt a folyamatot hívják dekódolásnak. Az azonosítók alapján minden komponens megtalálhatja a hozzá tartozó adatot, és frissítheti az esetlegesen csatolt modellobjektumot, gomb esetén pedig elindíthat egy action eseményt, amit majd a navigáció felhasználhat.

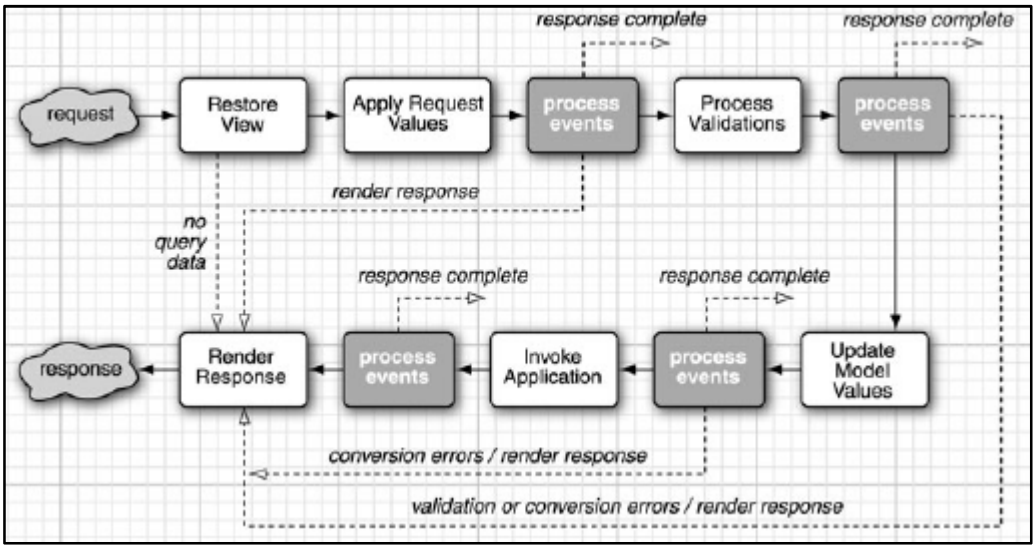
2.6. JSF Életciklus

Az előző fejezetben kifejtett két feldolgozási lépésen kívül a JSF specifikáció hat különböző fázist definiál. Ez az életciklus a 2.12.-es ábrán látható, ahol a normál működést a sima vonal jelöli, míg a kivételes eseteket a szaggatott vonal.

A Restore View fázis visszatölti a komponensfát, azaz végrehajtódik a dekódolás, amennyiben az már létrejött egyszer, vagy újat hoz létre, ha először kerül megjelenítésre az oldal. A már korábban megjelenített oldal esetén az összes komponens a legutóbbi állapotába töltődik vissza. Amennyiben a lekérésben nincs adat a JSF a Render Response fázisban folytatja a végrehajtást és a köztes lépések kimaradnak (például ez történik, mikor először jön létre a komponensfa).

A következő lépés az Apply Request Values. Ekkor a JSF implementációk végiglépkednek a komponensfán, és az összes komponens kiválaszthatja melyik lekérési adat

tartozik hozzá, majd a kiválasztott adatokat eltárolhatja. Az itt keletkező események, például egy gomb megnyomása esetén, egy eseménysorban tárolódnak, amit majd a fázis lezárulása után feldolgozhatnak az azokra feliratkozott eseménykezelők.



2.12. ábra. JSF életciklus

A Process Validations fázisban először a felhasználó által a lekérdezésbe tárolt szöveges adatok kerülnek konvertálásra. Amennyiben adtunk meg validátorokat az oldalhoz, azok is ebben a fázisban futnak le. Ha ebben a fázisban hiba történik a Render Response fázis következik, ahol a hibaüzenetekkel együtt megjelenítésre kerülnek a felhasználó által beírt adatok is, így azok javíthatóak lesznek, és nem kell újra begépelni őket. A konvertálások és az értékek helyességének ellenőrzése után a JSF feltételezi, hogy a megadott adatok helyesek, így továbblép az Update Model Values fázisba.

Az Update Model Values fázisban a komponensekhez kötött bean-ek adattagjai kerülnek beállításra, azaz a tulajdonságokat beállító metódusok futnak le.

Az Invoke Application fázisban az űrlap elküldését aktiváló gomb vagy link `action` attribútuma által indukált metódusok kerülnek végrehajtásra. A metódus tetszőleges műveletek végrehajtása után egy eredmény Stringet szolgáltat, amit a navigáció-kezelő használ fel, a következő megjelenítendő oldal meghatározására.

Végül a Render Response fázis következik, amikor is a keletkezett választ kódolja a JSF és elküldi a kliensnek.

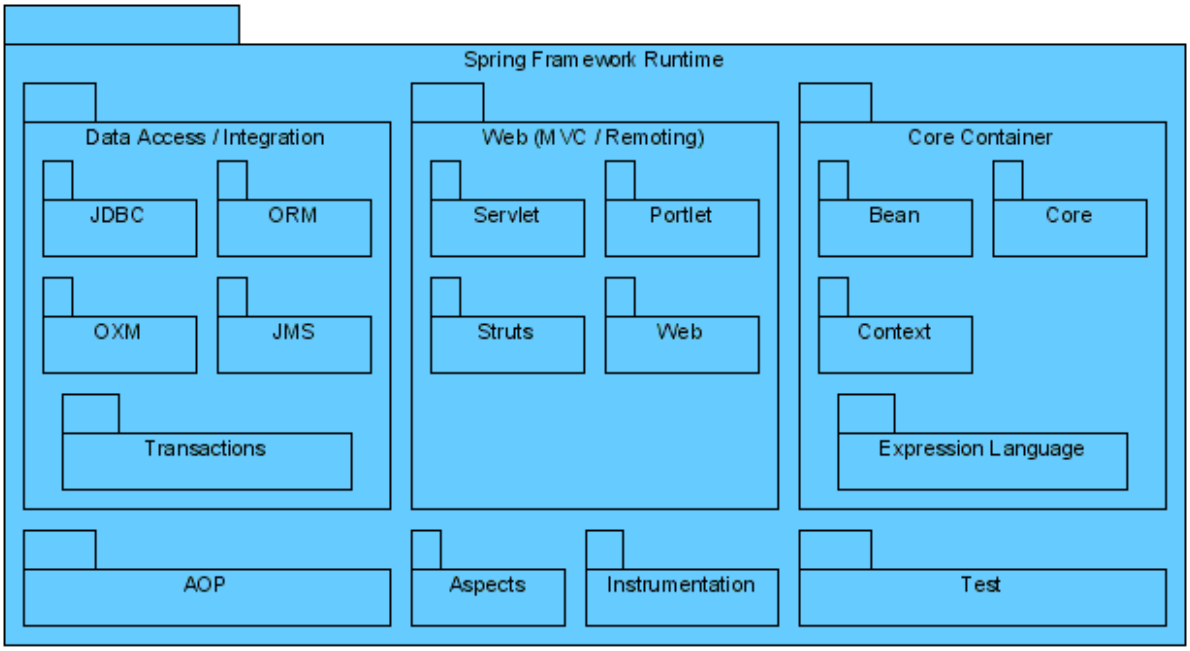
3. Spring Framework

A Spring először 2002 vége fele debütált Rod Johnson Expert One-on-One J2EE Design and Development könyvében, ahol a Spring mögött húzódó architektúrális gondolatok alapjai lettek lefektetve. Természetesen ezek az elképzelések már korábbian is megfogalmazódtak, így a gyökerek egészen a 2000-es évek elejéig nyúlnak vissza. Akkoriban, az internet lufi kidurranása néven elhíresült tőzsdei események után, elég kellemetlen helyzetben került a Java EE. Egyre másra jelentek meg a különböző keretrendszerek, melyek javítani próbálták a Java EE-ben történő fejlesztés nehézségeit több-kevesebb sikerrel. A Spring ebbe a környezetbe vezetett be egy frissítő új gondolkozásmódot, és 2003 januárjában a SourceForege, és az open source közösség támogatásával elindult a projekt.

A Spring alapjaiban arra lett tervezve, hogy webalkalmazásainkban POJO-kat (Plain Old Java Object) tudjunk használni úgy, hogy közbe ne kötelezzük el magunkat egy technológia mellett se. Ez a keretrendszer egy pehelysúlyú megoldás, de ezzel párhuzamosan nagy valószínűséggel nyújtja az összes olyan technológiát, amire majdnem minden fejlesztőnek szüksége van, miközben egyben moduláris is, így kihagyhatunk belőle minden olyan dolgot, amire nincs szükségünk. Úgy lett tervezve, hogy a vele fejlesztett alkalmazások a lehető legminimálisabban támaszkodjanak az API-jaira. Segítségével az EJB-k használata implementációs kérdéssé redukálódik, helyettük nyugodtan lehet használni akár POJO-kat is. További fontos előny a Spring használatával, hogy a vele fejlesztett alkalmazásokon nagyon egyszerű egység teszteket végezni.

Felmerülhet a fentiek olvasása után, hogy akkor pontosan mit is várhatunk el a Springtől? Habár a Spring sok területet magába foglal, a készítői pontosan tudják, és le is írják mire használható, és mire nem. A Spring alapvetően a Java EE-ben történő fejlesztés könnyítésére, és a jó programozási gyakorlatok alkalmazására fekteti a hangsúlyt a POJO-kkal való programozás lehetőségének biztosításával. Viszont a Spring nem találja fel újra a spanyolviaszt, így a csomagok között nem találhatunk naplózást (sajnálatos módon a tervezők vétettek egy hibát a Spring tervezésének legelején, és az egyetlen kötelező függőségnek a Commons Login (JCL) API-t tették meg, így a Spring jelenlegi verziója is a commons-logging csomagtól függ a kompatibilitás megőrzése miatt, amit elég problémás helyettesíteni), elosztott tranzakciókat, vagy connection pool-okat. Ezeket mind vagy nyílt forráskódú projektek biztosítják, vagy az alkalmazás szerver. Ugyanezen okból nincs objektum-relációs modell réteg sem, csak az azokat nyújtó technológiák támogatása, azaz a Spring a létező technológiák használatának a könnyítésére fekteti a hangsúlyt. Továbbá a Spring nem kíván direkt módon versenyezni más nyílt forráskódú projektekkel, csak akkor, ha a fejlesztő csapata szerint van lehetőség új dolgok felmutatására az adott területen belül. Ennek ellenére sok területen vált úttörővé a projekt. Fontos továbbá, hogy a Spring olyan előnyökből profitál, mint a belső konzisztencia, azaz az összes részprojekt azonos stílusban készül.

A Spring megközelítőleg 20 modulból áll (3.1. ábra). Ezek a modulok fő konténer, adatelérési, webes, AOP, eszközhasználat és teszt csoportokba lettek szétosztva. A fő konténerben lévő Bean és Core modulok az alapjai a Spring keretrendszernek, magukba foglalva az IoC konténert és a Dependency Injection szolgáltatásokat. Fő interfésze a BeanFactory, egy kifinomult megvalósítása a factory tervezési mintának, segítségével nincs szükség programjaink szintjén singleton-okat létrehozni, és könnyen elválaszthatjuk a konfigurációs és specifikációs függőségeket a programunk logikájától.



3.1. ábra. A Spring moduljai

A Context modul az előző két modulra épül és segítségével a Spring keretrendszer stílusában érhetjük el objektumainkat, hasonlít a JNDI (Java Naming and Directory Interface) nyilvántartáshoz. További szolgáltatásokat biztosít a Bean-ek mellé, mint az I18N, eseményszórás, valamint Java EE szolgáltatások támogatása, mint EJB, JMX és alapvető távoli szolgáltatások nyújtása. Alapja az ApplicationContext interfész.

Az adatelérési csomagok különböző integrációs rétegeket nyújtanak a JPA, JDO, Hibernate és egyéb ORM API-k számára, valamint absztrakciós réteget az objektum/XML leképezések számára, mint a JAXB, Castor, stb. A JDBC modul szintén absztrakciós réteget nyújt, amely elrejti az adatbázisfüggő hibaüzeneteket. Végül a JMS csomag üzenetek létrehozására és felhasználására tartalmaz szolgáltatásokat.

E szakdolgozat témájára vonatkozóan további nagyon lényeges modul a Web és a Web-Servlet, míg az előbbi web orientált szolgáltatások nyújtásáért, az IoC konténer servlet listener-ek segítségével való inicializálásáért és web orientált application context indításáért felelős, az utóbbi nyújtja az MVC tervezési minta webes implementációját.

3.1. IOC konténer

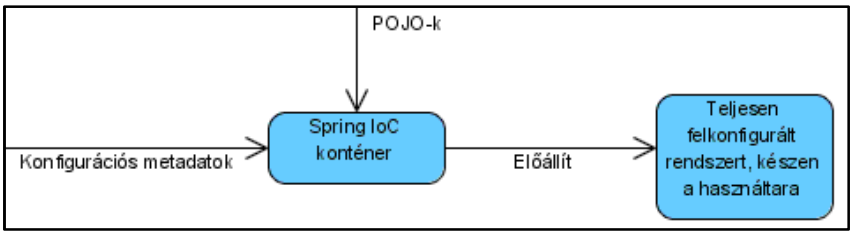
A Spring nem mással oldja meg a fent leírt rugalmasságot, mint az Inversion of Control és Dependency Injection fogalmak megvalósításával. Az IoC elvet gyakran a Hollywood stílussal szokták magyarázni: „Ne hívj minket, majd mi hívunk téged”. Olyan technikák széles tárházát takarja, melyek segítségével az objektumaink passzív résztvevői lesznek a rendszernek. Függőségeiket konstruktorban, gyártó metódusban megadott változókon keresztül vagy csak tulajdonságokkal határozzák meg, majd a keretrendszer pedig beléjük injektálja azokat a függőségeket, mikor létrejönnek ezek az objektumok.

A Dependency Injection viszont csupán az IoC egy konkrét megvalósítása, sokkal szűkebb fogalom, aminek a segítségével eltávolíthatók a konténer API-jaitól való explicit függőségek. Szokványos metódusok és konstruktorok lesznek felhasználva a függőségek beinjektálására, legyenek azok más objektumok, vagy csak egyszerű kezdeti értékek. Míg más architektúrákban a komponens jelez a keretrendszernek, hogy merre találja a szükséges objektumokat, addig itt a konténer számítja ki ezeket a függőségeket. A kiszámítás jelen esetben a metódusok szintaktikáján (a tulajdonságokhoz tartozó beállító metódusokén vagy konstruktorokén) és konfigurációs (például XML) adatokon alapul.

A bevezető részben említettekből kikövetkeztethető, hogy az IoC konténer alapjai az org.springframework.beans és org.springframework.context csomagokban találhatóak, és fő támpillérei a BeanFactory interfész és annak ApplicationContext alinterfésze. A BeanFactory szolgáltatja a konfigurációs keretrendszert, segítségével bármilyen típusú objektum kezelhető, és az alap funkcionalitást, míg az ApplicationContext további üzleti szolgáltatásokkal bővíti azt.

A Spring az IoC konténer által kezelt, az alkalmazás gerincét alkotó, objektumokat beaneknek hívja. A bean nem más, mint az IoC konténer által példányosított, összeállított vagy más módon kezelt objektum, és másképp megközelítve a bean egy az alkalmazást alkotó sok objektum közül. A beanek és a köztük lévő függőségek tükröződnek a konténer által használt konfigurációs metaadatokban.

Szóval az IoC-t reprezentáló ApplicationContext implementáció beolvassa a konfigurációs adatokat, azok segítségével példányosítja az objektumokat, majd beléjük injektálja a konfigurációban megadott függőségeket is, aminek végén egy teljesen felkonfigurált rendszert kaphatunk (3.2. ábra). A konfigurációkat sokféle képen megadhatjuk, a legelterjedtebb módjai az XML fájlok és Java annotációk, de történhet Java kódban is.



3.2. ábra. A Spring Core működése

Természetesen a Spring nem csak egy interfészeket biztosít, hanem ezen interfészek számos implementációját is. A készített projektben az XmlWebApplicationContext osztályt használom, ez a legelterjedtebb, legátláthatóbb és egyben ez jár a legkevesebb beállítási lépéssel is, mert alapbeállítás. Az XmlWebApplicationContext, mint a nevéből kikövetkeztethető XML konfigurációs fájlokból építi fel az objektumainkat és köztük lévő kapcsolatokat. Ha webes környezetben szeretnénk használni az IoC konténert, akkor a szükséges beállításokat a web.xml fájlban kell elvégeznünk. A konténer elindításáért, amennyiben használjuk a Spring MVC modult, a DispatcherServlet felelős, viszont ha nem szeretnénk használni az MVC nyújtotta szolgáltatásokat, akkor a Servlet 2.4-től kezdődően a ContextLoaderListener indítja azt (3.3. kódrészlet). A webalkalmazás gyökér application context-jének konfigurációját alapból a WEB-INF/applicationContext.xml fájlban keresi a rendszer, de ez átállítható a `contextConfigLocation` környezeti paraméter segítségével (3.4. kódrészlet).

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

3.3. kódrészlet

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring-config/root-application.xml</param-value>
</context-param>
```

3.4. kódrészlet

A bean-ek konfigurációját a `<bean>` elemekkel adhatjuk meg, egy `<beans>` gyökérelemen belül, a 3.5. kódrészleten látható sémakonfigurációval. Megfigyelhető, hogy a séma Spring 3.0.x verzióval készült.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="springBeanName" class="spring.bean.package.SpringBeanClass">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="anotherSpringBeanName" class="another.package.Class">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

</beans>
```

3.5. kódrészlet

Spring beaneknek tipikusan szolgáltatásrétegbeli objektumokat, adat elérést biztosító objektumokat (DAO), infrastrukturális objektumokat, JMS (Java Message Service)

üzenetsorokat és hasonló objektumokat szokás itt megadni. Az alkalmazás finomszemcsézett szakterületi objektumait tipikusan nem itt szokás definiálni, mert ezeket az üzleti logika hozza létre. Ha túl sok bean definíció kerülne egy fájlba és átláthatatlanná válik, van lehetőség több csoportra bontani az XML fájlt, majd a fő beállító fájlban az `<import resource=”xmlfajl.xml”>` elem segítségével beimportálni az alfájlokban található beandefiníciókat, alternatívaként a beanek egy részét érdemes lehet annotációkkal megadni. Az annotációkkal történő konfigurációt a `<context:annotation-config/>` elem segítségével kapcsolhatjuk be, miután definiáltuk a context XML névteret (3.6. kódrészlet). Következő lépés a `<context:component-scan base-package= "package.name"/>` megadása, ezzel definiáljuk ugyanis, melyik csomagot fésülje át az ApplicationContext az annotációk után.

```
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-3.0.xsd"
```

3.6. kódrészlet

A konténer használata nagyon egyszerű, webes környezetben pedig szinte sosem kerülünk vele konkrét kódbeli kapcsolatba, mert nem szükséges. Nem webes környezetben is egyszerűen kipróbálhatjuk a fenti beállításokkal rendelkező konténert, egyszerűen csak a ClassPathXmlApplicationContext osztálynak megadjuk paraméterként a konfigurációs fájlt, majd a `getBean` metódus segítségével elkérjük a konfigurált objektumot (3.7. kódrészlet).

```
// create and configure beans
ApplicationContext context = new ClassPathXmlApplicationContext(
        new String[] {"previous-config.xml"});

// retrieve configured instance
SpringBeanClass bean = context.getBean("springBeanName",
        SpringBeanClass.class);

// use configured instance
bean.doSomeMethodOrGetProperty();
```

3.7. kódrészlet

A konténeren belül a konfigurált beanek BeanDefiniton interfészt megvalósító osztályok objektumaiként lesznek reprezentálva. A bean létrehozásához felhasznált metaadatok a következők (ha nincs külön odaírva a `<bean>` tag attribútumairól van szó):

- Az osztály neve az azt tartalmazó csomaggal együtt (konstruktor inicializálás használata esetén), XML-ben a `class` attribútum.
- A bean neve, amivel majd hivatkozni tudjuk a konténerben, ajánlott a szokványos java elnevezési szabályokkal megadni. Ez az `id` attribútum XML-ben.
- A bean scope-ja, erről még később. XML-ben a `scope` attribútummal lehet megadni.

- A szükséges függőségek, amiket be kell majd injektálni az objektumba. Ez konstruktor argumentumok esetén a `<bean>` elemen belül a `<constructor-args>`, tulajdonságok esetén a `<property>` tag.
- Megadhatjuk, hogy a konténer a saját tartalmát átvizsgálva automatikusan döntse el az objektum szükséges függőségeit. Ezt az `autowire` attribútum beállításával tehetjük meg. Lehetséges értékei:
    - `no`: ekkor kikapcsoljuk a funkciót
    - `byName`: a tulajdonság neve alapján dől el a függőség, ha talál a konténerben a tulajdonság nevével rendelkező beant, akkor megpróbálja beinjektálni.
    - `byType`: A tulajdonság típusával rendelkező beant próbálja beinjektálni, ha több ugyanolyan típusú bean is található a konténerben kivétel keletkezik.
    - `constructor`: a `byType` csak konstruktorokra alkalmazva
    - `autodetect`: először `constructor` módban történik meg az injektálási kísérlet, ellenben ha van alapértelmezett konstruktor, akkor `byType` módban történik az injektálás.
- Lusta inicializáció is használható, ekkor a `singleton` scope-ban lévő beanek nem az application context indulásakor lesznek létrehozva, hanem az első hivatkozáskor. XML-ben a `lazy-init` attribútummal adható meg.
- A beanjeinkhez hozzárendelhetünk inicializáló és megsemmisítő metódusokat is, ezek a metódusok rendre a létrehozás után és a törlés előtt futnak le. XML-ben ezek az `init-method` és `destroy-method` attribútumokkal adhatóak meg (annotációk kezelése esetén működnek a megszokott `@PostConstruct` és a `@PreDestroy` annotációk is).

Az inicializáció többféle képpen is történhet, nem csak konstruktor segítségével. Amennyiben saját gyártó metódusaink vannak, vagy gyártó objektumunk, a kívánt objektumokat legyártathatjuk velük. Az objektumon belüli statikus gyártómetódus használatához a `class` attribútum helyett a `factory-method` attribútumot kell megadni a gyártó metódus nevével. Továbbá ha egy másik osztály példány szintű metódusát szeretnénk gyártó metódusnak használni, akkor előbb dekraláljuk bean-ként a gyártó objektumot, majd azt adjuk meg a gyártani kívánt bean `factory-bean` attribútumában, és a példány szintű gyártó metódust a `factory-method` attribútumban (3.8. kódrészlet).

```
<bean id="factoryObject" class="package.FactoryClassName"/>
<bean id="creatWithFactoryMethod" factory-bean="factoryObject" factory-
method="createThisBeanInstanceMethod"/>
```

3.8. kódrészlet

A függőségek beinjektálására vagyis két jelentős lehetőség van: a konstruktor alapú és a tulajdonság alapú (`<bean>` elemen belül `<constructor-arg>` vagy `<property>`). Mindkét esetben, ha osztályok példányait szeretnénk beinjektálni a `ref` attribútumot kell használni, míg konstans érték injektálására a `value` attribútummal van lehetőség.

Tulajdonságok injektálásnál mindig meg kell adni a tulajdonság nevét is. Konstruktor alapú injektálás esetén kicsit árnyaltabb a helyzet, ekkor ugyanis csak akkor nem kell explicit megadni a típusokat vagy sorrendet, ha paraméter listába az objektumok sorrendben és egyértelműen a megfelelő típussal rendelkeznek. Ellenkező esetben vagy a típust kell megadni plusz információként a `type` attribútummal, vagy a konstruktor argumentumának sorszámát az `index` (nullával kezdődően) attribútummal a `<constructor-arg>` elemben. Figyeljünk arra, hogy ha a konstruktorokban egymást körbehivatkozó beaneket adunk meg, azt az application context felismeri és BeanCurrentlyInCreationException kivételt fog kiváltani. Konfigurációkba értékként lehet még használni kollekciókat is, mint a List, Map, Set, Properties, és ezeket a definíciókat a különböző bean konfigurációkból egyesíteni is lehet.

A konfigurációs fájlban nem csak a függőségeiket és a konfigurációs értékeiket tudjuk definiálni az egyes beaneknek, hanem a hatásköreiket (scope) is. Az elérhető hatáskörök bővülnek attól függően, hogy milyen környezetben vagyunk. A két legalapvetőbb hatáskör a `singleton` és a `prototype`. Az első esetben az egész IoC konténerben csak egy példány kerül létrehozásra a beanből, míg a második esetben minden egyes hivatkozáskor egy-egy új példány jön létre. Amennyiben nem adunk meg `scope` attribútumot, alapértelmezettként a `singleton` hatáskörbe kerül az összes bean.

A további scope-ok csak akkor elérhetőek, ha webes környezetben vagyunk és ahhoz megfelelő application context implementációt használunk. Ezek eléréséhez csak akkor szükséges minimális beállítás, ha ismételten mellőzni szeretnénk az MVC modul szolgáltatásainak használatát, ugyanis a scopeok regisztrációját is a DispatcherServlet végzi. Az MVC használata nélkül ez a regisztráció a RequestContextListener feladata (3.9. kódrészlet).

```
<listener>
    <listener-class>
        org.springframework.web.context.request.RequestContextListener
    </listener-class>
</listener>
```

3.9. kódrészlet

A konfiguráció után a további hatáskörök a máshol már megszokott `request` és `session` scope-ok lesznek. A harmadik plusz scope a `globalSessio`n, de ez csak portál környezetben elérhető, így nem fejtem ki mit takar.

A hatáskörökkel szokásosan felmerülő kérdés, hogy hogyan tudunk tágabb scope-al rendelkező beanben szűkebb hatáskörű beant használni. A mindenki által gondolt válasz az, hogy egy konténerfüggő interfészt kell implementálni a nagyobb hatáskörű beannek, ami majd a kisebb hatáskörű aktuálisan létező beant mindig lekérdezi majd a konténertől. Ez ugyan lehetséges az ApplicationContextAware interfész segítségével, de a bevezetőben leírt célokkal, miszerint csak minél kevesebb explicit függés elfogadható a keretrendszertől, nem harmonizál. Ekkor lép be a képbe a metódusok injektálása, ugyanis a Springben nem csak

tulajdonságokat injektálhatunk, hanem újraimplementálhatunk bizonyos metódusokat is a konténer által kezelt beaneken. A metódusok beinjektálására futási időben kerül sor, bájtkódok generálásával. Egészen pontosan egy alosztály kerül generálásra a CGLIB könyvtár segítségével (azaz szükséges a CGLIB.jar, amihez pedig az ASM könyvtár), amiben felül lesz definiálva a kívánt metódus. A metódus injektálás egyik és itt kifejtett módja a `<bean>` elemen belüli `<lookup-method>`. Ezzel a módszerrel a publikus vagy protected, esetlegesen absztrakt, megfelelő visszatérési értékkel rendelkező paraméter nélküli metódusokat definiálhatjuk felül (3.10. kódrészlet a konfiguráció, 3.11. kódrészlet az osztály).

```
<bean name="prototypeBean" class="package.PrototypeBean"
    scope="prototype"/>
<bean id="methodLookupPresenter" class="package.MethodLookupPresenter"
    scope="singleton">
  <lookup-method name="getPrototypeBean" bean="prototypeBean"/>
</bean>
```

3.10. kódrészlet

```
public class MethodLookupPresenter{

    public void usePrototypeBean(){
        PrototypeBean pb = getPrototypeBean();
        pb.doSomeMethod();
    }

    // This could be abstract either.
    protected PrototypeBean getPrototypeBean(){
        return null;
    }
}
```

3.10. kódrészlet

Mint látható a Spring IoC konténere nagyon dinamikus, és rengeteg lehetőséget ismer, eszközeinek nagy része kiválóan konfigurálható. A fent leírt konfigurációknak már csak XML-en belül is több verziója létezik, ami ugyanarra az eredményre vezet, és az annotációkkal vagy kódban leírható, a fentiekkel analóg, beállítások nem is lettek leírva. További beállítási lehetőséget nem fogok kifejteni, csak felsorolás szintjén említeném meg, mert még oldalakon keresztül lehetne folytatni, és úgy érzem a legfontosabb beállítási lehetőségeket már ismertettem. Nem esett szó például a bean konfigurációk közötti öröklődésről, a beanek létrehozásakor esetlegesen fennálló időbeli függőségek megadásáról, az `idref` használatáról, névtelen beanek alkalmazásáról, az xml konfigurációs fájlok és az annotációk együttes alkalmazásának lehetőségéről, a beanekhez több hivatkozási név megadásáról vagy egyéni scope-ok létrehozásáról.

4. Spring Web MVC

A Spring MVC keretrendszer a DispatcherServlet köré épült fel. Ez az osztály a beérkező kéréseket szétosztja a megfelelő kontroller osztályoknak, amik majd feldolgozzák azt, miközben olyan különböző konfigurálható szolgáltatásokat nyújt, mint a megjelenítésért felelős eszköz kiválasztása, helyi nyelvi és stílus fájlok feloldása, vagy fájlfeltöltés támogatása. Segítségével, mint ahogy a Spring mag esetében is, bármilyen objektumot felhasználhatunk a kérések kezeléséhez, nincs szükség keretrendszerhez kapcsolódó interfészek implementálására. A megjelenítés is nagyon rugalmas, a kontrollerek akár a válaszüzenetbe közvetlenül is írhatnak, de tipikusan ModelAndView példányt adnak vissza. A ModelAndView objektumok gyakran a nézet nevét és egy Map-et tartalmaznak a nézeten megjeleníteni kívánt modellobjektumok számára. Mivel a modell csak a Map interfészen alapul, így nagyon könnyen integrálható, átalakítható és felhasználható bármely megjelenítésért felelős eszközben is.

A Spring MVC-vel való kapcsolat elején érdemes megjegyezni, hogy a „nyitott a bővítésre, de zárt a módosításra” elvek szerint tervezték, azaz bármit könnyedén hozzácsatolhatunk, de magát a rendszert csak nagyon kis mértékben módosíthatjuk. Ez nem okoz problémát, mert módosítás nélkül is nagyon használható a keretrendszer.

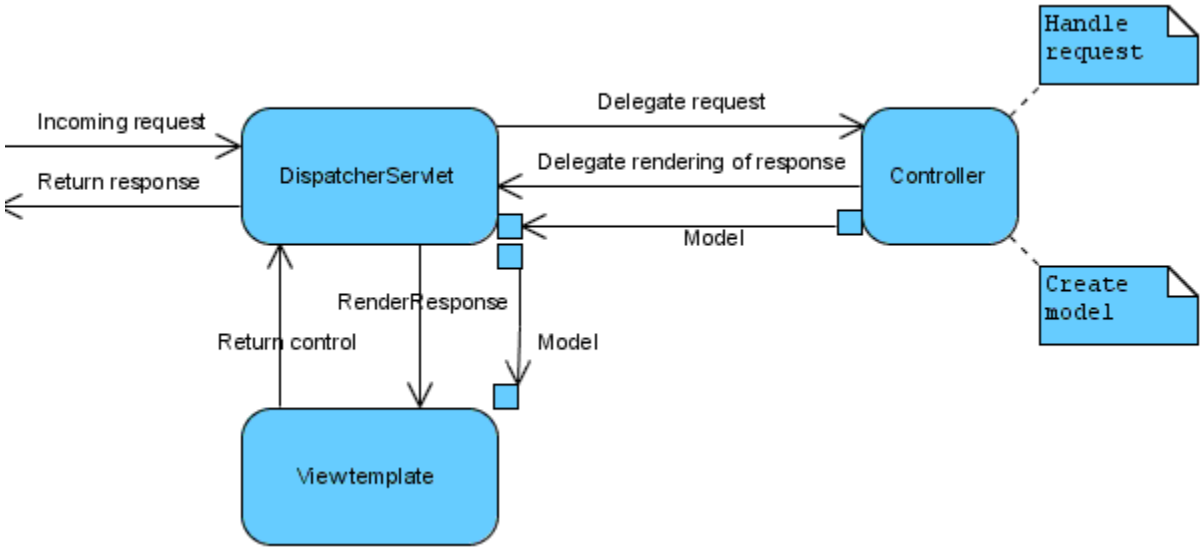
Az MVC által nyújtott szolgáltatások használatával nagy vonalakban a következő előnyökre lehet szert tenni:

- Szerepkörök tiszta elkülönítése (kontrollerek, a lekérést a kontrollerek kezelőmetódusainak kiosztó osztályok, modellobjektumok, konverterek, validátorok, stb.)
- A keretrendszer és az alkalmazás osztályainak egyszerű felkonfigurálása, mint JavaBean
- Rugalmas kontrollerdefiniálás
- Újrahasznosítható kódok
- Egyedi, igényekre szabható validáció és konverzáció
- Testre szabható a lekérések és kezelőmetódusok összerendelése és a megjelenítés
- Nyelvi és stílusfájlok használata
- Spring tag könyvtár melynek használata nem kötelező.
- JSP form tag könyvtár.

4.1. A DispatcherServlet osztály

Az Spring MVC alapja, mint fentebb említettem, a DispatcherServlet osztály. Mint sok más hasonló MVC keretrendszer a Springé is lekérdezés-vezérelt, azaz egy központi servlet köré épül, ami további kontroller osztályokhoz továbbítja a kéréseket (4.1. ábra). Ez a struktúra nem más, mint a FrontController tervezési minta megvalósítása. Az MVC

keretrendszer továbbá teljes mértékben profitál a Spring mag előnyeiből is, azaz az általa használt beaneket és beállításokat az ApplicationContext segítségével fogjuk megadni.



4.1. ábra. Spring MVC működésének folyamata

A DispatcherServlet osztály, mint a nevében is benne van, egy servlet, kiterjeszti a HttpServlet osztályt. Mint minden servlet-et, természetesen a web.xml fájlban kell konfigurálni, a tetszőleges URL kötés megadásával (4.2. kódrészlet). Amennyiben korábban nem adtuk meg a ContextLoaderListener-t, akkor a gyökér ApplicationContext-et a DispatcherServlet fogja elindítani a megadott beállítások szerint. Mivel webes környezetben vagyunk, minden egyes DispatcherServerlet-hez tartozik egy külön ApplicationContext is, ami a gyökérből származik. A gyökérben definiált összes bean öröklődik a servlet környezetébe. Az öröklődött beaneket tetszőlegesen felüldefiniálhatjuk, ezek az új bean definíciók a szűkebb környezetre lesznek érvényesek. Alapértelmezett esetben a servlet környezetének a beállításait a [servlet-neve]-servlet.xml fájlban kell megadni. Ha ezt át szeretnénk állítani egy másik fájlra, akkor a konfigurációs fájl elérési útját a DispatcherServlet beállításainál tehetjük meg a contextConfigLocation paraméter segítségével (4.2. kódrészlet).

```
<servlet>
  <servlet-name>Dispatcher Servlet</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring-config/mvc.xml</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>Dispatcher Servlet</servlet-name>
  <url-pattern>/spring/*</url-pattern>
</servlet-mapping>
```

4.2. kódrészlet

A beállításokat a megfelelő konfigurációs fájl segítségével végezhetjük el, és a következő beaneket definiálhatjuk (A legtöbb beannek egyébként nem szükséges azonosítót adnunk, mert a DispatcherServlet végigböngészi a környezetet, és az adott interfészt kiterjesztő beaneket automatikusan regisztrálja.):

- `multipartResolver`: fájlok feltöltéséhez használható HTML űrlapokon keresztül.
- `localeResolver`: segítségével a kliens által használt lokális nyelvi beállításokat tudjuk kideríteni, I18N-hez szükséges.
- `themeResolver`: A stílusok használatához adhatjuk meg, így létre lehet hozni akár személyre szabott stílusokat is.
- `handlerMapping`: Több kezelő-hozzárendelőt is beállíthatunk, segítségükkel a beérkező kéréseket oszthatjuk ki a megfelelő kontroller kezelőmetódusának. Működésekor végrehajtja a kontrollert megelőző interceptorokat, a kontrollert magát, és végül a rákövetkező interceptorokat is.
- `handlerAdapter`: A kiválasztott kontroller kezelőmetódusát hajtja végre, amennyiben az támogatott. Mivel a kontrollerek csak az Object osztály leszármazottjai ezért saját HandlerAdapter írásával bármilyen, a kérést kezelni képes, már működő kódot integrálhatunk az MVC-be (mondjuk más keretrendszerből). Nem az alkalmazásfejlesztőket célozták meg, mikor létrehozták ezt az interfészt, azaz ezt inkább csak más keretrendszerrel való együttműködés megvalósítására kell használni.
- `handlerExceptionResolver`: segítségével a keletkezett kivételekhez rendelhetünk nézeteket, vagy komplex hibakezelő kódokat írhatunk.
- `viewNameTranslator`: Az explicit hozzárendelés nélküli logikai nézetek kezeléséhez adhatunk meg stratégiát.
- `viewResolver`: A logikai nézetek nevét rendeli megjeleníthető nézetekhez.

A rendszer tetszőleges felkonfigurálása után a működés több lépésben írható le. Elsőként a kérés a beérkezés után lementésre kerül a WebApplicationContext-be a `DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE` kulccsal. Ezek után a `localResolver`, a `themeResolver` majd a `multipartResolver` lesz a lekéréshez kapcsolva, amennyiben voltak ilyen eszközök megadva. Következő lépésként a környezetből kiválasztásra kerül a megfelelő kezelő-hozzárendelő és az általa visszaadott kontroller kezelőmetódusa végrehajtásra kerül. Végül a kezelőmetódus által adott eredmény megjelenítésre kerül. Ha a kezelőmetódus nem adott vissza modellt, akkor a megjelenítés nem történik meg, ekkor ugyanis már a válaszüzenet nagy valószínűséggel elkészült. A kérés feldolgozása közben keletkező kivételeket a WebApplicationContext-ben deklarált kivételkezelők kapják meg.

4.2. A kezelő-hozzárendelők

A Spring 2.5.-ös verzió előtt szükséges volt HandlerMappings interfészt megvalósító osztály megadására a beanek között, viszont a 2.5.-ös verziótól kezdődően csak akkor szükséges konfigurálnunk ilyen beant, ha szeretnénk valamilyen általánostól eltérő viselkedést megadni. A fent említett interfészt implementáló bean nélkül a DispatcherServlet automatikusan bekonfigurálja a DefaultAnnotationHandlerMapping osztályt, ami a kontrollereinkben megadott `@RequestMapping` annotációkat fogja értelmezni (erről később még lesz szó).

Vagyis kezelő-hozzárendelő bean létrehozására csak akkor van szükség, ha szeretnénk megadni a kontroller lefutása előtt vagy után működésbe lépő interceptorokat, alapértelmezett kezelő-hozzárendelőt, vagy kezelő-hozzárendelők közötti sorrendet (4.3. kódrészlet), esetleg az `alwaysUseFullPath`, `urlDecode`, `lazyInitHandlers` logikai tulajdonságok alapértelmezett beállításain akarunk változtatni.

A legegyszerűbb az annotációkat felismerő kezelő-hozzárendelők használata, de további más kezelő-hozzárendelők közül is választhatunk. Minden kezelő-hozzárendelő a beérkező kérés URL-je alapján dönti el, hogy melyik kontroller melyik kezelőmetódusát fogja választani, a hozzá regisztráltak közül. Az egyik legtöbb beállítási lehetőséget a SimpleUrlHandlerMapping biztosítja, segítségével konkrét URL-hez vagy URL-mintákhoz rendelhetünk tetszőleges kontrollereket (4.3. kódrészlet). A többi kezelő-hozzárendelő kevesebb beállítási lehetőséggel rendelkezik, használatuk épp ezért egyszerűbb is. Például a BeanNameUrlHandlerMapping a beanekhez megadott becenevek (alias) alapján rendel az URL-ekhez kontrollereket (csak akkor ha /-el kezdődik). Itt is használható minta, amennyiben az `alwaysUseFullPath` hamis, különben csak pontos egyezés alapján kerül kiosztásra a kontroller. További kezelő-hozzárendelők a ControllerBeanNameHandlerMapping és a ControllerClassNameHandlerMapping osztályok, amik vagy a konfigurált beanek nevéhez vagy a beanek osztályának nevéhez hasonlítják az URL-t, és ez alapján határozzák meg a kontrollert.

```
<bean class=
        "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="interceptors">
    <list><ref bean="interceptAny"/></list>
  </property>
  <property name="mappings">
    <props>
      <prop key="/flow/*">flowController</prop>
      <prop key="/logout">logoutController</prop>
    </props>
  </property>
  <property name="defaultHandler">
    <bean class=
        "org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
  </property>
</bean>
```

4.3. kódrészlet

A kontrollerek lefutása előtt és után végrehajtható kódot az interceptorok segítségével szúrhatunk be. Az interceptor nem más, mint egy HandlerInterceptor interfészt implementáló osztály, amit definiáltunk beanként és regisztráltunk a kívánt kezelő-hozzárendelőben. Az előbb említett interfész egyébként három metódust definiál. A `preHandle` metódus a kontroller előtt fut le, a `postHandle` az után, míg az `afterCompletion` a kérés teljes végrehajtása után kerül meghívásra. Ha nincs szükségünk mind a három metódusra, és gyakran nincs, akkor nem szükséges mindet implementálni, hanem csak a HandlerInterceptorAdapter osztályt kell kiterjeszteni. Ez az osztály ugyanis implementálja a HandlerInterceptor interfészt üres metódustörzsekkel.

4.3. A kontrollerek

A kontrollerek az alkalmazás viselkedéséhez nyújtanak hozzáférést, amiket tipikusan szolgáltatások interfészein keresztül szoktak definiálni. A kontrollerek értelmezik a felhasználó által beírt adatokat, majd azokat modellé transzformálják. Ezek a modellek aztán a nézetek segítségével megjelenítődnek a felhasználónak. A Spring a kontrollereket erősen absztrakt módon implementálja, így könnyedén kreálhatunk sommindenből kontrollert.

Kontrollerek megadásának legegyszerűbb és legrugalmasabb módja az annotációk használata. A megfelelő, annotációkkal definiált beanek osztályait tartalmazó, csomag megadása után (IoC konténer fejezetben leírt módon) a `@Controller` annotációval jelezhetjük, hogy az adott osztály egy kontroller. Annotációk használata esetén nincs szükség arra, hogy bármilyen Spring interfészt implementáljuk.

A kontroller osztályok kezelőmetódusokból állnak. Hogy ezek a metódusok működésbe lépjenek a `@RequesMappings` annotációt is meg kell adni, amennyiben az alapértelmezett DefaultAnnotationHandlerMapping osztályt használjuk kezelő-hozzárendelőnek, ekkor ugyanis a DispatcherServlet automatikusan átböngészi a kontroller osztályokat és felépíti a kezelőmetódusokhoz tartozó URL-mintákat. A `@RequestMapping` annotációval az egész kontroller osztályt vagy csak annak metódusait rendelhetjük URL-ekhez. Például az 4.4. kódrészletben, ha a felhasználó a …/spring/mvc címre lett irányítva, akkor a `setupMvc` metódus fog lefutni (csak a GET kérések esetén, mert a kontroller tovább lett finomítva metódus szinten). Figyeljünk rá, hogy ha felüldefiniáljuk az alapértelmezett beállításokat, akkor a metódus szintű `@RequestMapping` annotációk működéséhez az AnnotationMethodHandlerAdapter beant is explicit definiálnunk kell!

```
@Controller
@RequestMapping(value="/mvc")
public class MvcController {
	@RequestMapping(method = RequestMethod.GET)
	public String setupMvc( ModelMap map ){
		map.addAttribute("hello", "Hello World!" );
		return "viewName";
	}
}
```

4.4. kódrészlet

Az osztály szinten megadott leképezések az egész osztályra vonatkoznak, és az osztályon belül lehet tovább finomítani egyéb `@RequestMapping` beállításokkal, így megadva azt, hogy a lekérést melyik metódus kezelje le. Ez az annotáció egyébként négy paramétert definiál, amik mind tömbök. A `value` segítségével adhatjuk meg azokat az URL mintákat, amik észlelése esetén a kontroller osztályunk kezelje a lekérést. Lényeges hogy Spring már ismeri az úgynevezett URI Template-eket is, így adhatunk meg {} zárójelek között nevesített mintát is, amire később hivatkozhatunk a `@PathVariable` annotációval (4.5. kódrészlet). A `method` paraméter segítségével a beérkező kérés http metódusa szerint szűkíthetjük a kört. Végül a `header` és a `param` segítségével a lekérdezés fejlécében vagy paramétereiben előforduló értékekre írhatunk ki megszorításokat (például csak akkor fusson le a kezelőmetódus, ha a lekérés tartalmaz egy bizonyos paramétert, vagy csak szöveges lekérdezésekre lépjen működésbe stb.).

```
@RequestMapping(value="/{templateName}")
public String method(
    @PathVariable(value="templateName") String renamedTemplate,
    Model model
){…}
```

4.5. kódrészlet

A `@RequestMapping` annotációk elég széleskörű metódusszignatúrára értelmezhetőek. A kezelőmetódusok paraméterei a következőek lehetnek. Ezeket a paramétereket egyébként majd futáskor a rendszer injektálja be:

- Request vagy response objektumok (pl ServletRequest, HttpServletResponse).
- Session objektumok (pl HttpSession).
- WebRequest vagy NativeWebRequest.
- Locale objektumok.
- InputStream/Reader és OutputStream/Writer
- `@PathVariable`, `@RequestParam`, `@RequestHeader`, `@RequestBody` annotációkkal ellátott metódusparaméterek.
- Map, Model, ModelMap objektumok a nézet számára.
- Command vagy form objektumok.
- Errors/BindingResult
- SessionStatus

Visszatérési érték esetén pedig:

- ModelAndView, korábban már említett objektum a nézet nevével és a modellel
- Model, ekkor a nézet implicit határozódik meg.
- Map, szintén modellt tartalmazza, ekkor is implicit kerül meghatározásra a nézet.
- View, ekkor a modell kerül implicit meghatározásra.
- String, a logikai nézet nevével.

- void, ha a kontroller maga felelős a válasz generálásáért, vagy explicit meghatározható a nézet.
- Egyéb `@ModelAttribute` metódusszintű annotációval dekralált visszatérési érték.

Mivel a szakdolgozat nem az MVC működéséről szól, ezért a további lehetőségekről csak nagyvonalakban írok. Egyszóval a `@RequestMapping` annotációval azt írhatjuk le, hogy a kontroller melyik kezelőmetódusa fusson le a beérkező kérések URI-je, fejléce, http metódusa vagy paraméterei alapján. A `@PathVariable` annotációt, mint már említettem, az URI Template-ek használatára alkalmazhatjuk a kezelőmetódus paramétereinek megadásakor. A `@RequestParam` segítségével a kérés paramétereit érhetjük el szintén a metódus paramétereinél, míg a `@RequestBody`-val a kérés törzsét, a `@CookieValue`-val a http sütik értékeit rendelhetjük paraméterekhez, végül a `@RequestHeader` annotációval pedig a kérés fejlécét használhatjuk fel. A `@ResponseBody` annotációval ellátott metódus esetén a visszatérési érték rögtön a válasz törzsébe íródik. Végül a `@ModelAttribute` annotációt a metódus paraméterlistájában alkalmazva a modellből rendelhetünk attribútumokat a paraméterekhez, metódus szinten használva pedig a visszatérési értéket helyezhetjük el a modellbe.

4.4. A nézetek

Mint minden MVC keretrendszer, a Spring is nyújt a modell megjelenítésére eszközöket. A Spring a nélkül teszi ezt, hogy bármilyen megjelenítési technikához kötné magát, de ettől függetlenül azonnal használhatunk több ilyen technikát is, például JSP-ket, Velocity sablonokat vagy XSLT nézeteket. A Spring MVC megjelenítésért felelős része a View és a ViewResolver interfészekre alapszik.

Minden kezelőmetódus meghatároz explicit vagy implicit egy logikai nézet nevét, ami majd megjeleníti a modellt. A logikai nézetek nevét a ViewResolver fogja fizikai nézetekhez kötni úgy, hogy létrehoz egy View objektumot, majd visszaadja azt megjelenítésre a DispatcherServlet-nek. A nézetek feloldására a következő osztályokat konfigurálhatjuk az ApplicationContext-ben:

- AbstractChacingViewResolver: A legtöbb nézetfeloldó kiterjeszti ezt az osztály, segítségével a már egyszer megjelenítésre került nézetek tárolva lesznek, így nem kell azokat újra feldolgozni.
- XmlViewResolver: XML konfigurációs fájl alapján oldja fel a nézeteket.
- ResourceBundleViewResolver: hasonló az előzőhöz csak a konfigurációt resource fájlból nyeri.
- UrlBasedViewResolver: A leggyakrabban használt nézetfeloldó, azonnal URL-re fordítja a logikai nézetek nevét, ebből következik, hogy a nézeteink logikai neveinek meg kel egyezniük a fizikaival. Az általa visszaadott nézetek osztályát megadhatjuk a `viewClass` paraméterrel.

- InternalResourceViewResolver: az előző osztály leszármazottja, InternalResourceView nézeteket hoz létre, amiket JSP-k és Tiles-ok megjelenítésére használhatunk.

Például JSP-k használatához a ViewResolver konfigurálása az 4.6. kódrészleten látható. A „helloWorld” logikai nézet nevét lefordítja a /WEB-INF/jsp/helloWorld.jsp URL címre, és ezt adja vissza megjelenítésre.

```
<bean id="jpsViewResolver" class=
        "org.springframework.web.servlet.view.UrlBasedViewResolver">
  <property name="viewClass" value=
        "org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

4.6. kódrészlet

A nézetfeloldókat láncba is lehet kötni, így ha az egyik nem tud nézetet rendelni a logikai névhez, akkor a következő nézetfeloldó kerül meghívásra. Láncot létrehozni egyszerűen több ViewResolver definiálásával lehet a környezetben. Meghatározni, melyik feloldó kerüljön előrébb a láncba, az `order` tulajdonság megadásával lehet. Ha nem adtunk meg a sorrendre vonatkozó tulajdonságot, akkor az ilyen nézetfeloldó a lista végére fog kerülni. A nézetek működése direkt úgy lett definiálva, hogy a visszatérési érték lehet `null` is, ekkor ugyanis a nézetet nem sikerült megtalálnia az adott ViewResolver-nek, így a lehetőség a listában lévő következő feloldóra száll. Amennyiben a folyamat végén egyetlen ViewResolver-nek sem sikerült feloldania a logikai nézet nevét, akkor váltódik ki kivétel. Fontos tudni, hogy néhány ViewResolver minden esetben létrehoz View objektumot, így az ilyen feloldókat mindig a lista végére kell helyezni!

A Spring MVC lehetőséget biztosít a Post-Redirect-Get minta megvalósítására is, ugyanis bizonyos esetekben fontos, mint mikor http post után az egyik kontroller elvégezte a dolgát és átadná az irányítást egy másiknak, hogy a nézet ne azonnal kerüljön megjelenítésre. Ez főleg a böngésző frissítésekor, vagy a vissza gomb használatakor lényeges, hogy ne kerüljön elküldésre újra a korábban már elküldött post adat. A redirect-et kikényszeríteni a logikai névben megadott „redirect:” prefix segítségével lehetséges, amennyiben az UrlBasedViewResolver-t vagy annak alosztályát használjuk. Az ilyen nézetfeloldó ugyanis mindig megvizsgálja a logikai nézet nevét, és amennyiben ezzel a stringel kezdődik, akkor egy RedirectView osztályt ad vissza, ami meghívja megjelenítéskor a HttpServletResponse.sendRedirect() metódust. Ha nem UrlBasedViewResolver osztályt használunk, mint nézetfeloldó, a redirectet a kontroller önmaga kényszerítheti ki a RedirectView objektum létrehozásával és visszaadásával. A fentiekhez hasonló módon használható forwardolás is UrlBasedViewResolver osztály használatával a „forward:” prefix segítségével.

A modell megjelenítéséért a nézetfeloldó által visszaadott View interfész implementáció felel. Ezek az implementációk elég eltérőek lehetnek egymástól. A Spring MVC alapból elég sok View implementációt biztosít, többek között excel, pdf, xslt, json és különböző jsp technológiák megjelenítésére. A View implementációnak mindig állapot nélküli beannek kell lennie és szálbiztosnak. Például a JSP-k megjelenítésére az InternalResourceView és alosztályai szolgálnak. Ezek a JSP-k megjelenítésért felelős osztályok a modellobjektumokat `request` attribútumokként teszik elérhetővé és a megadott erőforrásra mutató URL-el továbbítják a beérkezett kérést a RequestDispatcher osztály segítségével.

4.5. JSF integrációs lehetőségek

A JSF integrációs lehetőségek több szintből állhatnak. A legerőteljesebb integrációt a Spring Web Flow használata jelenti, segítségével a JSF eszközei szinte teljes mértékben lecserélésre kerülnek, lényegében csak a megjelenítésre lesz felhasználva, erről bővebben majd a következő fejezetben írok. Ennél gyengébb integrációs lehetőségként a JSF Spring MVC-vel történő használata jelenti. Ekkor a JSF szintén, mint megjelenítő eszköz lesz felhasználva csak.

Az MVC-vel való integrációra a Spring Faces csomagban találhatunk egy nézetet generáló osztályt, az org.springframework.faces.mvc.JsfView-t. Aki egy pillantást vet erre az osztályra, láthatja hogy a View interfészt implementálja, azaz a konfigurációban megadható a view feloldón belül mint `viewClass`. A JsfView osztályban található `renderMergedOutputModel()` metódus az, ami majd a JSF nézetek megjelenítését fogja elvégezni. Ez az osztály szinte semmi pluszszolgáltatást nem nyújt (még a hibaüzenetek se kerülnek konvertálásra, viszont a modellt el lehet érni a JSF nézeteken), csak gyorsan végigmegy a JSF életciklusán, majd megjelöli elkészültnek a FacesContextet, azaz az itt renderelt nézet fog megjelenni a kliensnél.

Az utolsó integrációs lehetőség a JSF oldalról való közelítés. Ebben az esetben a JSF beanekben és beállítási fájl(ok)ban szeretnénk elérhetővé tenni a WebApplicationContext használatát. Ezt az 1.2.-es JSF verziótól a SpringBeanFacesELResolver osztály segítségével tehetjük meg. Miután definiáltuk a faces-context.xml-ben mint el-resolver (4.7. kódrészlet), az EL kifejezéseket előbb a Spring környezetnek delegálja, majd ha nem talált megfelelő objektumot a JSF környezetben keresi azt tovább.

```
<application>
  <el-resolver>
    org.springframework.web.jsf.el.SpringBeanFacesELResolver
  </el-resolver>
</application>
```

4.7. kódrészlet

A korábbi, JSF 1.1-es, verziókban a DelegatingVariableResolver és a SpringBeanVariableResolver osztályok használhatóak, mint variable-resolver-ek. Az előbbi

először a JSF környezetben keresi az objektumokat, majd a Spring környezetet fésüli át, míg az utóbbi pont fordítva, előbb a Spring környezete jön, és csak utána a JSF-é.

5. Spring Web Flow

Amikor elkezdünk egy honlapot tervezni, legtöbbször egy funkcionális és viselkedési modell kerül felrajzolásra, ahol a cselekvések különböző lépéseit vizuálisan meg tudjuk jeleníteni. Ezek a diagramok nagyon hasonlóak az UML állapotdiagramjaihoz. Az ilyen állapotdiagramokban minden állapot egy kezdő, inicializáló állapotból indul ki, majd különböző események szerint elkezd állapotot váltani. Az állapotokhoz megadhatunk különböző tevékenységeket, amik elvégzésre kerülnek, valamint definiálhatnak állapot-kezdő és -vég tevékenységeket is. Az állapotok között nyilak a lehetséges állapotátmeneteket jelölik, ezek akkor következhetnek be, ha az őrző feltétel igaz. Az UML állapotdiagramok tartalmazhatnak szuper állapotokat is. Ezek akkor kerülnek használatra, mikor több állapot is ugyanazt az állapotátmenetet írja le, így ahelyett hogy többször leírnánk ugyanazt az állapotsorozatot, inkább egy szuperállapotot hozunk létre, amit több helyen is felhasználhatunk. A diagram állapotátmeneteinek sorozata, amennyiben nem tartalmaz végtelen ciklust a diagram, mindig eléri a végállapotot. Hogy most miért is volt szó az UML állapotdiagramról? Nem másért, mint a véges állapotú gép miatt. Az UML állapotdiagramjai ugyanis ezt írják le, továbbá a Spring Web Flow is annak egy implementációja.

A Spring Web Flow létrejöttének oka főleg az oldalak közötti navigáció leírására vezethető vissza. További okként szerepel a servlet specifikáció azon hiányossága is, hogy a hatáskörök nem biztosítanak elég rugalmasságot. Azaz a legtöbb esetben a `request` hatáskör túl rövid, míg a `session` túl hosszú. Az előzőekből rengeteg probléma adódhat, mint erőforrás pazarlás, névtér ütközés és modell-implementációk közötti jelentős eltérések. Így az SWF azért jött létre hogy a lehető legjobb navigációt biztosítsa.

Előnyei közé tartozik, hogy több hatáskörrel is kibővíti a servlet specifikációt, azaz előtérbe lépnek a flowk, mint a párbeszédes scopeok megvalósítói. A különböző flowk párhuzamosan működhetnek a nélkül, hogy bármiféle befolyással lennének egymásra, és a párbeszéd végén a felhasznált erőforrások automatikusan felszabadításra kerülnek. Az SWF teljes mértékben absztrahált a servlet specifikációtól. Ez azt jelenti, hogy az SWF-ben flowkról nézetekről, állapotokról és akciókról van szó. Egy flow leírhat pár weboldalt vagy akár pár GUI ablakot is. Kezelése nagyon könnyű, és átlátható, nem programozói beállítottságú emberek is könnyen megérthetik a vizuálisan leírt flowkat.

További előnyök közé tartozik, hogy a flow nem több mint egy MVC kontroller. Ez azt jelenti, hogy több kontrollert is használhatunk szabadon. Még ha a projektünk egy részére az SWF-et is választottuk, a többi részére egyáltalán nem szükséges azt használnunk. Tervezésekor a legjobb gyakorlatokra támaszkodtak, azaz teljes mértékben interfész alapú, így mindig a leghelyénvalóbb implementációt használhatjuk. Valamint a finomszemcsézettségű eseményfigyelővel további beállítási lehetőségekhez juthatunk. Az SWF nagyban támaszkodik az MVC-re és a Spring magra, azok jó tulajdonságait megtartva, így nagyjából könnyen összeintegrálhatjuk már létező kódokkal, csak tudnunk kell, épp

milyen adaptert kell írnunk. Végül a flowk tesztelhetősége, mint a Springé általában, nagyon könnyű és hatékony.

Az SWF-nek nem csak előnyei vannak, hanem mint minden rendszernek, hátrányai is. Legfontosabb ezek között, hogy az SWF-et inkább több oldalt tartalmazó, valamilyen logikát végigvezető esetekben érdemes használni. Ez azért lehet kellemetlen, mert a JSF integrációval, nehézkes lesz kezelni egyoldalas modelleket, és az MVC-vel való alap integráció nem nyújt kellő használhatóságot. Ez esetben csak simán a JSF-et érdemes használni a Spring magjának támogatásával. További hátrányok közé tartozik az SWF szegényes referenciája, és a róla készült irodalom vékonysága. A fórumokon feltett kérdésekre se mindig szokott válasz érkezni, így ütközhetünk megoldatlan problémákba is (gyakran nem az SWF hibájáról van szó), igaz ezeket legtöbb esetben valamilyen kevésbé elegáns kerülő úton meg lehet oldani.

5.1. Konfiguráció

A Spring Web Flow egy MVC kontroller, mint előzőekben már említettem, így konfigurációja is a DispatcherServlet környezetében történik. Erre két fő lehetőség van. Első a FlowHandlerMapping és a FlowHandlerAdapter konfigurálása (5.1. kódrészlet).

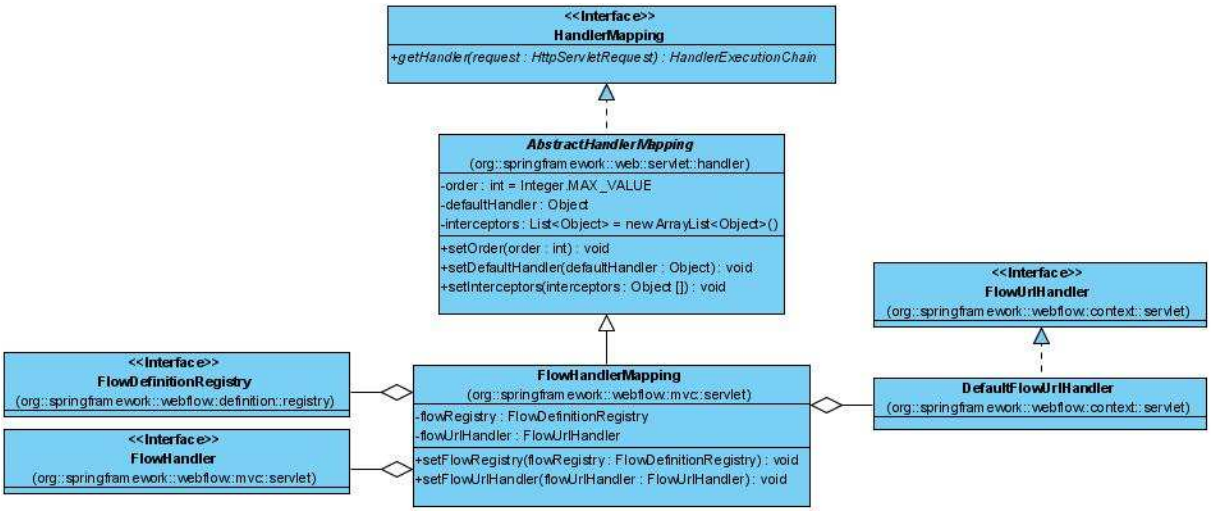
```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
  <property name="flowRegistry" ref="flowRegistry"/>
</bean>

<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
  <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

5.1. kódrészlet

A FlowHandlerMapping (5.2. ábra) alapból a DefaultFlowUrlHandler osztályt használja a beérkező címek feloldására. Ez a leképező egyszerűen csak az aktuálisan (FlowRegistry-ben) regisztrált flowk azonosítójára próbálja lefordítani a beérkező kérések URL címét. Például a /spring/some-flow URL esetén a some-flow azonosítóval rendelkező flowt próbálja megtalálni. Más leképezési stratégiát is alkalmazhatunk a FilenameFlowUrlHandler osztály beállításával, vagy saját FlowUrlHandler implementációval. Amennyiben a FlowHandlerMapping nem talál megfelelő flowt a nyilvántartásában, akkor null-t ad vissza, hogy az MVC további kezelő-hozzárendelőire kerülhessen a sor.

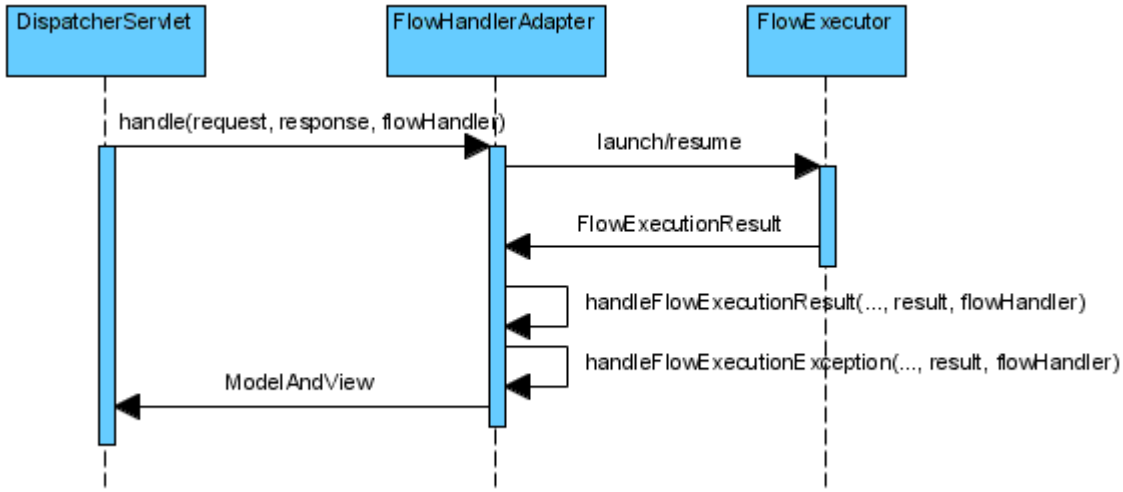
A kiválasztott flow indítását a FlowHandler interfészt implementáló osztály segítségével szabhatjuk testre. A FlowHandlerMapping használata esetén az ilyen osztályokat a flow azonosítójával beanként kell definiálni. Ebből következik, hogy minden egyes flowhoz külön FlowHandler implementációt kapcsolhatunk. Ha explicit ezt nem tesszük meg, akkor az AbstractFlowHandler osztályt kibővítő, csak a flow azonosítót kezelő alapértelmezett osztály lesz kiosztva az aktuális flownak. Az ilyen segítőosztályok használatával három dolgot befolyásolhatunk:

- Testre szabhatjuk a flow indításakor a modell bemeneti paramétereit.
- Kezelhetjük a flow befejeződésekor esetlegesen keletkező eredményt.
- A flowban keletkező, el nem kapott kivételeket kezelhetjük tetszés szerint.



5.2. ábra. FlowHandlerMapping osztályszerkezete

A FlowHandlerAdapter foglalja magába a flowk végrehajtásával kapcsolatos utasítások láncát, mint a flowk indítását, folytatását és a visszatérési eredmény kezelését (5.3. ábra). A flow indítása és folytatása olyan tevékenységek, amelyek a végrehajtónak, azaz a FlowExecutor-nak lesznek delegálva, és majd FlowExecutionResult objektummal térnek vissza. Ezen objektum alapján fog majd eldőlni, hogy milyen redirect történik. A flow működése alatt bekövetkező kivételeket is és a flow működésének befejeztével keletkező eredményeket is a FlowHandlerAdapter adja át az aktuális flowhoz tartozó FlowHandler-nek.



5.3. ábra. FlowHandlerAdapter működésének főbb lépései

Másik megközelítés az SWF és az MVC integrálására a FlowController használata. Egy ilyen beállítást már leírtam az 4.3. kódrészletben. Ott a SimpleUrlHandlerMapping osztály segítségével lett kiosztva, hogy a megadott URL-re érkező kéréseket a FlowController

hajtsa végre, azaz ebben az esetben nem az SWF kezelő-hozzárendelőjét használjuk, hanem az MVC-ben már beállítottra hagyatkozunk. Ilyen esetben a FlowController-t is beanként kell deklarálni, mint minden más kontrollert (5.4. kódrészlet). A FlowController nagy részben az Adapter tervezési minta megvalósítása, és a Spring MVC kontrollereként ugyan azt a tevékenységet végzi, mint az előző beállítások mellett a FlowHandlerAdapter. Ebből kifolyólag a FlowController-ben megadott FlowHandlerAdapter-nek lesznek delegálva a lekérések, és az összes FlowHandlerAdapter-ben megadható tulajdonságokat itt is beállíthatjuk. Ha explicit nem állítjuk be ezt a belső használatra szánt adapter osztályt, a kontroller automatikusan konfigurál egyet magának. Mivel ezzel a kontrolleres beállításokkal nincs FlowHandlerMapping definiálva, a kontroller átveszi a FlowHandler interfészt implementáló segítőosztályok kezelését.

```
<bean id="flowController" class=
        "org.springframework.webflow.mvc.servlet.FlowController">
  <property name="flowExecutor" ref="flowExecutor"/>
  <property name="flowHandlers">
    <map>
      <entry key="myFolwId" value-ref="myFlowHandlerImpl"/>
    </map>
  </property>
</bean>
```

5.4. kódrészlet

A flowk indításáért és folytatásáért a FlowExecutor felelős. Ez az interfész az SWF egyik központi eleme. Összesen csak két metódust foglal magába: a launchExecution-t és a resumeExecution-t. Csak ez a két metódus rejti el a flowk végrehajtásának a belső bonyolultságát. Az alapértelmezett implementáció a flowk létrehozására, indítására és folytatására három segítőosztályt használ:

- FlowDefinitionLocator: fő feladata a flowk definícióinak visszaadása azonosító alapján. Ezt az interfészt tovább bővíti a FlowDefinitionRegistry, és ezek implementációja, a FlowDefinitionRegistryImpl osztály tartalmazza az összes ApplicationContext-ben regisztrált flow definícióját.
- FlowExecutionFactory: Ez az interfész felelős a FlowExecution objektumok összeállításáért a kívánt FlowDefinition példányokból, valamint a meghatározott eseménykezelők regisztrálásáért is a létrehozott végrehajtási objektumokban.
- FlowExecutionRepository: Felelős a flowk perzisztens kezeléséért. Feladatai közé tartozik a flowk mentése, visszaállítása és törlése a repository-ból. Minden egyes repository-ba elmentett FlowExecution objektum egy adott flow állapotát tükrözi egy adott pillanatban. Minden ilyen flow állapot egy egyedi végrehajtási kulccsal rendelkezik, aminek neve flowExecutionKey. Ennek a kulcsnak a segítségével történhet a flow állapotának visszanyerése egy későbbi időpontban.

FlowExecutor regisztrálására a `webflow:flow-executor` elem segítségével történhet (az `xmlns:webflow="http://www.springframework.org/schema/`

webflow-config" névtérben). Mivel a végrehajtó e módszerrel történő definiálása a FlowExecutorFactoryBean segítségével történik, az abban az osztályban található tulajdonságokat is itt adhatjuk meg. Azaz eseménykezelőket a további `webflow:flow-execution-listeners`, különböző flowal kapcsolatos attribútumokat a `webflow:flow-execution-attributes` tagon belül adhatunk meg. Szabályozhatjuk az egy felhasználó által létrehozható flowk, és az egyes flowk tárolásra kerülő állapotainak számát is a `webflow:flow-execution-repository` tag segítségével. (5.5. kódrészlet)

```
<webflow:flow-executor id="flowExecutor" flow-registry="flowRegistry">
  <webflow:flow-execution-repository max-execution-snapshots="10"
                                     max-executions="5"/>
  <webflow:flow-execution-listeners>
    <webflow:listener ref="securityFlowExecutionListener" />
    <webflow:listener ref="jpaFlowExecutionListener"/>
  </webflow:flow-execution-listeners>
</webflow:flow-executor>
```

5.5. kódrészlet

Ahhoz hogy a létrehozott flowink kiválaszthatók legyenek végrehajtásra, regisztrálnunk kell azokat a `webflow:flow-registry` elem segítségével. A flowk definícióit egyenként vagy minta szerint is megadhatjuk. Előbbi esetben a `webflow:flow-location` utóbbiban a `webflow:flow-location-pattern` elemeket kell használnunk. (5.6. kódrészlet)

```
<webflow:flow-registry id="flowRegistry"
                       flow-builder-services="customFlowBuilderServices">
  <webflow:flow-location path="/MyFlowDir/another-flow.xml"/>
  <webflow:flow-location-pattern value="/WEB-INF/flows/**/*-flow.xml"/>
</webflow:flow-registry>
```

5.6. kódrészlet

A `webflow:flow-registry` tag a FlowRegistryFactoryBean osztályt használja a FlowRegistry felépítésére. Mint látható az 5.5. kódrészleten ebben az elemben beállíthatunk saját flowépítő szolgáltatásokat, amik segítségével befolyásolhatjuk a flowk építését. Három segítőosztály állítható be: a ConversionService, ExpressionParser és a ViewFactoryCreator (5.7. kódrészlet).

```
<webflow:flow-builder-services id="customFlowBuilderServices"
                      conversion-service="myConversionService"
                      expression-parser="myExpressionParser"
                      view-factory-creator="myViewFactoryCreator"/>
```

5.7. kódrészlet

A ConversionService leszármazott osztályai tárolják a flowk végrehajtása közben szükséges konvertereket. Alapértelmezett esetben több stringből konvertáló osztály be lesz állítva, ezek a tipikus java típusokra konvertálnak, mint a boolean, int, date, satöbbi. Saját konvertáló megadására is itt van lehetőség, legegyszerűbb módon a DefaultConversionService osztály kibővítésével. Saját konverterünknek, egyirányú

konvertálás esetén a Converter-t, kétirányú esetén a TwoWayConverter interfészt kell kiterjesztenie, de erről pontosabban majd később.

Az ExpressionParser segítségével az SWF működése közbeni kifejezések elemzését szabhatjuk testre. Jelenleg az SWF csak két EL könyvtárat támogat. Alapértelmezett a JBoss, a másik pedig az OGNL EL könyvtár.

Végül a ViewFactoryCreator interfész a megjelenítéshez szükséges gyárat hozza létre. Két implementációja van az SWF-be beépítve. A JsfViewFactoryCreator a JSF integrációjához, és az MvcViewFactoryCreator az MVC-ben már használt nézetfeloldók használatához.

A fenti beállítások részleteiből nem biztos, hogy összeállt az olvasó számára a flow működése, így kifejtem most egészben az első beállítási verzióval. Mikor a kérés beérkezik a DispatcherServlet-hez, az a FlowHandlerMapping segítségével kiválasztja a megfelelő FlowHandler-t, legtöbbször a flow azonosítója alapján, majd a FlowHandlerAdapter segítségével kezeli azt. A FlowHandlerAdapter először a lekérés paraméterei között keresi a flow végrehajtására utaló paramétereket, ha nincsenek ilyen paraméterek, akkor a FlowExecutor segítségével elindul egy új flow végrehajtása, azaz egy új FlowExecution. A FlowExecutor új flow indításakor a flow definícióját, azaz a FlowDefinition-t, a FlowDefinitionLocator segítségével szerzi meg. Az így kinyert FlowDefinition-t aztán a FlowExecutionFactory kapja meg, és használja fel a FlowExecution-ök gyártására. A FlowExecutor mielőtt visszaadná a FlowExecutionResult-ot az új végrehajtás objektumot elmenti a FlowExecutionRepository-ba. Amennyiben a FlowHandlerAdapter megtalálja a megfelelő paramétereket, azok felhasználásával visszanyeri a FlowExecutinRepository-ból a megfelelő FlowExecution objektumot, aminek a végrehajtása aztán folytatódik. A FlowExecution futása közben létrejött új állapota is tárolódik a FlowExecutionRepository-ban a flow végrehajtási eredményének meghatározása előtt. Végül a FlowExecutionResult objektum, amely a FlowExecution adott vissza a FlowHandlerAdapter-nek, a FlowHandler esetleges felhasználásával, meghatározza az új megjelenítendő nézetet.

5.2. Flow-k definiálása

Flow-k definiálása főleg XML segítségével történhet. Minden a flowt leíró állapotot az 5.8. kódrészleten látható gyökérelemben kell felvenni. Mint már írtam, a definiált flowkat hogy használhassuk, meg kell adni vagy minták segítségével vagy konkrét elérési úttal a FlowRegistry-nek.

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
    http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

</flow>
```

5.8. kódrészlet

A flow lényegében egy véges állapotú gép, ahol az állapotok különböző viselkedéseket hajtanak végre, és mivel ezeknek a végrehajtható állapotoknak a viselkedése nagyban eltérhet egymástól, az SWF öt állapot típust különböztet meg.

A leggyakrabban használt állapot a `view-state`. Segítségével megjeleníthetjük a webalkalmazásunk kimenetét, így a felhasználó aktívan részt vehet a flow folyásában. Minimálisan egy flown belül egyedi azonosítót kell megadnunk. Definíció szerint ekkor az azonosító neve lesz felhasználva a nézet megjelenítésére. A nézetet a flowt definiáló fájlhoz relatívan kell megadni, az előbbi esetben azonos könyvtárban. Nézetet explicit a `view` attribútum segítségével adhatunk meg három féle módon: relatívan a webalkalmazás környezetének gyökeréhez képest / jellel kezdve, relatívan a flow definíciós fájlhoz képest csak a nézet nevének megadásával (5.9. kódrészlet), valamint logikai nézetek neveit is megadhatjuk, amit a ViewResolver old majd fel. A nézetek nevében megadhatunk speciális értékeket is, amiket az `externalRedirect:` direktívával kell kezdeni:

- `servletRelative`: a jelenlegi servlethez képest megadott erőforrásra irányít
- `contextRelative`: a jelenlegi webalkalmazáshoz képest megadott erőforrásra irányít
- `serverRelative`: a szerver gyökeréhez képest megadott erőforrásra irányít
- `http://` vagy `https://`: teljesen megadott URL alapján történik a redirect.

```
<view-state id="viewID" view="viewName.xhtml">
</view-state>
```

5.9. kódrészlet

A döntési állapot abban az esetben jöhet hasznosan, ha valamilyen futási idejű döntés alapján kell állapotot váltani a flowban (5.10. kódrészlet). Lehetőség van egy kifejezés kiértékelésére, aminek visszatérési értéke dönti el, melyik állapotátmenet hajtódjon végre. Az `else` attribútum nem kötelező, elhagyása esetén több `if` tag is megadható. Amennyiben nem az utolsó `if` elemben van az `else` attribútum, akkor a hátralévő `if` elemek nem fognak végrehajtódni, mert az állapotátmenet korábban fog bekövetkezni. A tesztelő metódusban bármilyen EL kifejezés megadható, de óvakodjunk az üzleti logika végrehajtásától itt, mert ez az állapot a flow állapotainak irányításáért lett létrehozva.

```
<decision-state id="decideAboutDetails">
  <if test="objectInScope.moreDetailsNeededParam"
      then="moreDetailsNeededView" else="noMoreDetailsNeededView"/>
</decision-state>
```

5.10. kódrészlet

A flowk végét az `end-state` állapottal adhatjuk meg (5.11. kódrészlet). Ha a legfelső szinten lévő flow elér egy ilyen állapotba, akkor a végrehajtása befejeződik, és azt nem lehet újra fojtatni. Ha egy al-flow ér el egy ilyen állapotba, akkor a szülő flow folytatódik, felhasználva a befejeződött flow eredményét. Amennyiben máshogy nem rendelkezünk ez az eredmény az `end-state` azonosítója lesz. Alapból a végállapot nem

jelenít meg nézeteket, ez a viselkedés megfelelő alflowk esetén. Viszont legfelső szintű flowknál megadhatjuk milyen nézettel folytatódjon tovább a webalkalmazásunk. Ilyenkor jól jöhetnek a nézetek nevében megadható direktívák.

```
<end-state id="end" view="lastView.xhtml"/>
```

5.11. kódrészlet

Alflowkat a `sub-flow` elemmel indíthatunk (5.12. kódrészlet). Ekkor egy új flow jön létre, mint a szülő flow alflowja. Az indító flow várakozó állapotba kerül, amíg az alflow végrehajtása be nem fejeződik. Mikor az alfow befejeződik, újra a szülőflowba kerül az irányítás, és a befejezett flow lehetséges eredménye befolyásolhatja a szülőflow következő állapotátmenetét.

```
<subflow-state id="stateId" subflow="startedSubflowId">
  <input name="inputAttributeName" value="scopeObject"/>
  <transition on="outcomeOne" to="viewOne"/>
  <transition on="outcomeTwo" to="viewTwo"/>
</subflow-state>
```

5.12. kódrészlet

Végül az utolsó állapot az `action-state` (5.13. kódrészlet). Ebben az állapotban különböző üzleti logikát lehet végrehajtani, majd a logika által adott visszatérési értékek alapján tovább haladni a következő állapotba. Ezt az állapotot a SWF 2.0-ban már nem szükséges használni, mert helyette az `<evaluate>` elem segítségével szinte bárhol végrehajthatunk tetszőleges logikát biztosító metódusokat.

```
<action-state id="actionId">
  <evaluate expression="serviceBean.doLogic()" result="scope.attribute"/>
  <evaluate expression="otherServiceBean.doMoreLogic(attribute)"
            result="scope.otherAttributeName"/>
  <transition to="viewState"/>
</action-state>
```

5.13. kódrészlet

A flow állapotai közötti átmenetek kiváltására a `<transition>` elem való. Ezt az elemet a végállapoton kívül bármely állapotban használhatjuk. Az események bekövetkeztekor sorban összehasonlításra kerülnek az átmenetek nevei, és az első egyezés alapján történik meg az állapotátmenet (5.14. kódrészlet). Amennyiben `view-state`-ben nem adunk meg célállapotot az átmenetben, akkor ugyan az a nézet fog újrarenderelődik.

```
<view-state id="stateID" view="viewName.xhtml">
  <transition on="eventOne" to="anotherViewStateID"/>
  <transition on="eventTwo" to="someActionStateID"/>
  <transition on="eventThree"/>
</view-state>
```

5.14. kódrészlet

Ha a flow több olyan állapotot is tartalmaz, amiben ugyanolyan átmenetek is találhatóak, akkor használható a `<global-transition>` állapoton kívüli elem. Ezzel olyan átmeneteket definiálunk, amely a flow minden egyes állapotára érvényes lesz.

Az állapotok közti átmeneteket az események indítják el. Egy esemény alapvetően az állapot végrehajtásának kimeneti értéke. Ez lehet egy metódus visszatérési értéke, vagy a felhasználó által a nézeten indított esemény, mint a gombokra vagy linkekre való kattintás. Eseményt kiváltani a megjelenítési technikától függően GET metódus esetén legtöbbször az `_eventId` paraméter és utána az esemény nevének lekérésben visszaküldésével lehet, míg POST metódus esetén `form` és `input` elemek együttes használatával két féle képen is. Egyik módszer az `input` nevének `_eventId`-t adni, értékének az esemény nevét, másik az `input` nevének `_eventId_${esemény_neve}` paramétert adni. JSF használata esetén az események ugyanúgy következnek be, mint a normális JSF navigáció estében.

Az adatmodell eléréséhez, beanek metódusainak hívásához, és változók futás időbeni kiértékeléséhez a Spring Web Flow is az Expression Language-et használja, a korábban már írt két implementáció segítségével. Amennyiben a jboss-el könyvtárat és az el-api könyvtárat elérhetővé tesszük az elérési útban, az SWF automatikusan ezt fogja használni. Az EL használatával az SWF következő dolgokra lesz képes:

- Kifejezések feloldására és kiértékelésére, mint például a nézetek nevei és állapotátmenetek kritériumai.
- Hozzáférésre a kliens által küldött adatokhoz, flow attribútumok és lekérési paraméterek segítségével.
- Hozzáférésre szerver oldali adatstruktúrákhoz, mint a Spring ApplicationContext objektumaihoz vagy a flow által definiált hatáskörökhöz.
- Metódusok meghívására Spring beaneken.

A Spring Web Flow két féle kifejezést különböztet meg. Az egyik a simán feloldani kívánt kifejezés, a másik pedig a kiértékelni kívánt kifejezés. Előbbi esetben használni kell az EL határolójeleit, mint a `#{}` vagy a `${}`, míg utóbbi esetben ezt el kell hagyni. Kiértékelendő kifejezés megadására az `evaluate` elem használandó (5.15. kódrészlet). Kifejezéseket a névtérben elérhető beaneken végezhetünk el az `expression` attribútum használatával. Ha a metódus visszatérési értékét később szeretnénk felhasználni, és nem pedig a navigációhoz, tárolhatjuk azt valamelyik scope-ba (használható scope-okról később). A visszatérési értéket az elérhető konverterek segítségével tetszőleges típusra konvertálhatjuk. Az elérhető konvertereket a korábban a flow beállításainál megadott ConversionService tartalmazza.

```
<evaluate expression="bean.method()" result="flowScope.resultName"
          result-type="explicitResultType"/>
```

5.15. kódrészlet

Az SWF a flow életciklusának több pontján is engedélyezi a kiértékelni kívánt kifejezések, azaz akciók, hívását, minden egyes ponton az `evaluate` elem használatával. Ezek a pontok a flow indulása (`<on-start>` tag), a flow vége (`<on-end>`), állapot elindulása (`<on-entry>`), állapot vége (`<on-exit>`), nézet megjelenítése előtt (`<on-render>`) és végül minden `<transition>` elemen belül is.

A flowknak, akár legfelső szintűek, akár csak al-flowk, lehetnek bemenő paraméterei, valamint visszaadhatnak kimenő paramétereket, mikor futásuk véget ér. Ez akkor lehet hasznos, ha a flow a lekérés adatain manipulál, vagy az al-flow egy működése közben létrehozott objektumot szeretne visszaadni. Elvárt bemeneti paramétert a flow definíciójának elején az `<input>` elem segítségével adhatunk meg. Az `input` tag `name` attribútuma, mint kulcs szerepel, ezen a néven lesz elérhető a bemeneti paraméter a flowban, a `value` attribútum pedig a bemeneti értéket jelenti. További beállításként megadhatjuk, hogy a megfelelő kulcsú argumentumnak kötelezően léteznie kell a flow indulásakor. Lényeges megemlíteni, hogy referencia nem adható át a `value` attribútum segítségével, ekkor a `name` attribútumnak meg kell egyeznie az átadni kívánt korábban már létrehozott paraméter nevével (azaz így nem nevezhetjük át). Kimenő paramétert a végállapotban adhatunk meg az `<output>` elem segítségével.

5.3. JSF integráció - Spring Faces

A Spring Faces modul biztosítja a JSF integrációját. Ez a csomag tartalmazza azokat az implementált osztályokat, amik segítségével megjeleníthetőek a JSF nézetek. Segítségével összeolvaszthatóak a JSF erősségei (főleg az UI komponensek), a Spring erősségeivel, elhagyva azok gyengeségeit. A Spring Faces továbbá biztosít egy apró Facelet komponenskönyvtárat is, amely AJAX és kliens oldali validációs lehetőségeket nyújt. Ez a komponenskönyvtár a Spring JavaScript csomagra épül, ami a Dojo nevezetű JS eszközkönyvtárt integráló absztrakciós JS keretrendszer.

A Spring Faces használatával a következő előnyöket élvezhetjük: a Spring által kezelt beanek, scope-ok kezelése, eseménykezelés, navigációs szabályok, egyszerű modularizálás, nézetek csomagolása, letisztult URL-ek használata, modell szintű validáció, kliens oldali validáció, és részleges oldalfrissítések AJAX-al. Ezekkel az előnyökkel, kevesebb beállítás mellett tisztábban elkülönülő nézet és kontroller rétegeket kapunk, az alkalmazásunk funkcionalitásának sokkal tisztább moduláris felbontása mellett.

JSF integrációjához a web.xml-ben definiálnunk kell a JSF FacesServlet osztálya mellett a Spring DispatceherServlet-ét is. Ha szeretnénk használni a Spring Faces komponenseit is, szükségünk lesz a Spring JS ResourceServlet-ére is (5.16. kódrészlet), ami a SF komponensekhez szükséges CSS és JS erőforrásokat biztosítja akár .jar fájlokból is. Ezeknek az erőforrásoknak a kérései mindig a /resources/* címre érkeznek. Az SF komponensek a facelet technológiára épülnek, ezért biztosítanunk kell a Facelets kiegészítő

könyvtárat is az elérési útban, az ahhoz elengedhetetlen beállítások megadása mellett a web.xml-ben (5.17. kódrészlet) és a faces-config.xml-ben (5.18. kódrészlet).

```
<servlet>
  <servlet-name>Resources Servlet</servlet-name>
  <servlet-class>
    org.springframework.js.resource.ResourceServlet
  </servlet-class>
  <load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Resources Servlet</servlet-name>
  <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

5.16. kódrészlet

```
<context-param>
  <param-name>facelets.VIEW_MAPPINGS</param-name>
  <param-value>*.xhtml</param-value>
</context-param>
```

5.17. kódrészlet

```
<application>
  <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
</application>
```

5.18. kódrészlet

A Spring Faces komponenseinek használatához egy lehetséges .XHTML névtérbeállítás az 5.19. kódrészleten látható, szokványos esetben ez az `sf`, valamint a faceletek-nek az `ui`. Az `includeStyles` és az `includeScripts` segítségével a Spring Faces komponensek stílusait és a működésükhöz szükséges JavaScript-eket adhatjuk meg. Ezeket a fájlokat majd a ResourceServlet használatán keresztül fogja a kliens letölteni. A dokumentáció ajánlása szerint, az optimális működés érdekében mindig az oldal fej részében adjuk meg ezeket a komponenseket (5.20. kódrészlet). Az `includeStyles` esetében nem szükséges a `themePath` és a `theme` attribútumok használata, alapértelmezett esetben a tundra stílus lesz használva, ami az org.springframework.js.jar fájlban található meg, pár másik előre definiált stílus mellett.

```
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:sf="http://www.springframework.org/tags/faces"
```

5.19. kódrészlet

```
<head>
  <sf:includeStyles themePath="/dijit/themes/" theme="nihilo"/>
  <sf:includeScripts />
</head>
```

5.20. kódrészlet

A publikálást leíró és a JSF konfigurációs fájl beállítása után a következő lépés a Spring Web Flow beállítása a JSF nézetek kezeléséhez. Ez a beállítás a `xmlns:faces`

="http://www.springframework.org/schema/faces" névtér megadása után a webflow konfigurációs fájljában az 5.21. kódrészleten látható `flow-builder-services` használatával lehetséges.

```
<faces:flow-builder-services id="facesFlowBuilderServices"/>
```

5.21. kódrészlet

A `flow-builder-services` segítségével több dolgot is testre szabhatunk, mint azt már korábban leírtam. A Spring Faces által definiált elemeket használva a következő alapértelmezett beállítások kerülnek kiosztásra, a flowk létrehozását segítő osztályokként:

- `conversion-service`: a FacesConversionService osztály lesz beállítva, mely a Web Flow-ban alapértelmezettekhez további egy konvertert ad hozzá. A plusz egy konverter a JSF táblázatainak kezeléséhez használható DataModel objektumok konvertálását biztosítja.
- `expression-parser`: a WebFlowELExpressionParser osztályt állítja be, amely a flow definíciókban található EL kifejezéseket oldja fel.
- `enable-managed-beans`: alapértelmezettben hamis, ha átállítjuk igazra a WebFlowELExpressionParser osztály helyett a JsfManagedBeanAwareELExpressionParser osztály lesz beállítva az EL kifejezések feloldására. Ez az osztály lehetővé teszi a JSF kezelte bean-ek elérését a flowkban.
- `view-factory-creator`: a JsfViewFactoryCreator osztályt állítja be a nézetek létrehozásához. Ez minden egyes nézethez létrehoz egy JsfViewFactory osztályt, ami majd kezeli a JSF életciklus első lépését.

Most hogy a Spring Web Flow-t beállítottuk, bemutatok pár Spring Faces komponenst is. Az UICommand komponensek alapból AJAX alapokon működnek, ez a viselkedés a komponens `ajaxEnabled` attribútumával állítható át. Amennyiben a kliens nem képes az AJAX kezelésére, ez nem fog semmi gondot okozni, ugyanis a komponenseken kiváltott események ekkor teljes oldalfrissülést eredményeznek.

Az `includeStyles` és `includeScripts` elemeket már említettem, de ezeken kívül van még további lehetőség a ResourceServlet-en keresztül erőforrások elérésére, méghozzá a `resource` és `resourceGroup` komponensek használatával. A Spring Faces biztosít pár kliens oldali validátort is. Ezen validátorok használata kicsit trükkös, ugyanis nem a használni kívánt komponensen belül kell megadni őket, hanem fordítva, bennük kell megadni a komponenst. Validálni az `inputText` mezőt lehet a `clientTextValidator`, `clientNumberValidator`, `clientCurrencyValidator` és `clientDateValidator` segítségével (az utolsóra példa az 5.22. ábra az 5.23. kódrészlettel, ami az alapértelmezett stílusú dátumokat validáló komponens). További validáló eszköz a `validateAllOnClick`, ami az összes kliens oldali validáló eszközt elsüti.

5.22. ábra. AZ SF ClientDateValidator komponense

```
<sf:clientDateValidator required="true" invalidMessage="Wrong date">
  <h:inputText id="checkinDate" value="" required="true">
    <f:convertDateTime pattern="yyyy.MM.dd" timeZone="EST"/>
  </h:inputText>
</sf:clientDateValidator>
```

5.23. kódrészlet

Az oldalak részleges frissítését a `commandButton` és a `commandLink` komponensekkel lehet kiváltani. Az előbbi két komponens szokványos JSF attribútumain kívül a `processIds` a leglényegesebb, amelyben megadhatjuk mely kliens oldali DOM elemek frissüljenek a gomb megnyomása után. Hogy a szerver oldalon is részleges legyen a frissítés, azt a `view-state` elemeken belül a `render` tag segítségével állíthatjuk be. Ezen elem `fragments` attribútumával ugyanis a frissíteni kívánt JSF komponensek részfájának gyökerét adhatjuk meg. Ez az elem nem AJAX kérések esetén figyelmen kívül lesz hagyva, és a teljes oldal újrarenderelődik. Van egy további lehetőség részleges oldalfrissítés kiváltására az `ajaxEvent` komponens segítségével. Ezt a komponens teljesen JavaScript alapú így csak akkor szabad használni, ha az általa kiváltott esemény elmaradása nem okoz komoly gondot az alkalmazás működésében. Csak három attribútuma van, a korábban már kifejtett `processIds`, az `action` és az `event`. Az `event` segítségével megadhatjuk, hogy az alkomponens melyik DOM eseményének kiváltódása esetén küldje el a szervernek az `action` attribútummal megadott JSF eseményt.

6. Az SWF és a JSF 2.0 összehasonlítása

6.1. Navigáció

A JSF 2.0-ban az 1.2-ben már megszokott navigáción túl bevezetésre került pár új hasznos eszköz is. Az egyik ilyen az implicit navigáció lehetősége. Amikor a navigáció-kezelő végigvizsgálta az összes navigációs szabályt és nem találta meg az adott eseményhez a megfelelő nézetet, akkor a JSF 2.0-ban történik még egy vizsgálat arra vonatkozóan, létezik-e az esemény nevével egy nézet. Ha létezik, akkor az a nézet kerül megjelenítésre. Ezzel a módszerrel jelentősen lecsökkenthető a faces-config.xml navigációs szabályainak a mennyisége, ugyanis a statikus navigáció esetén a gomb `action` attribútumában a következő oldal nevét adhatjuk meg, dinamikus esetben pedig a metódus visszatérési éréke lehet a nézet neve. Implicit navigáció segítségével is kiválthatunk redirectet a `faces-redirect=true` paraméter nézet nevében történő megadásával (6.1. kódrészlet).

```
public String buttonAction(){
  return "nextPage?faces-redirect=true";
}
```

6.1. kódrészlet

A következő újdonság a feltételes navigáció. Feltételes navigációval, az `if` tag segítségével, megadhatunk olyan feltételt, aminek teljesülnie kell, mielőtt az aktuális navigációs eset kiválasztásra kerülne. Mindezt az üzleti logikától vagy a beanektől szétválasztva tehetjük meg ezzel a módszerrel, így ezeknek nem kell navigációtól függő kimeneteket előállítaniuk. (6.2. kódrészlet)

```
<navigation-rule>
  <from-view-id>/currentPage.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>nextPage</from-outcome>
    <if>#{testBean.ifConditionParameter}</if>
    <to-view-id>/nextPage.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

6.2. kódrészlet

Végül a navigációban történt változtatások utolsó nagyobb része a NavigatonHandler interfészt implementáló osztállyal kapcsolatos. A JSF 2.0-ban bevezettek egy új alinterfészt, a ConfigurationNavigationHandler-t, és az új referenciaimplementáció már ezt az interfészt valósítja meg. Az ebben definiált pluszszolgáltatásokkal hozzáférhetünk a navigációt tartalmazó adatokhoz, így kideríthetjük, hogy bizonyos kimenetek megadása milyen eredményre vezet, továbbá egyszerűbben kiválthatunk navigációt a performNavigation metódus segítségével.

Ezzel ellentétben a Spring Web Flow-ban implicit navigációra nincs lehetőség, legalábbis nem olyan formában, mint azt az előbb a JSF 2.0-ban láthattuk. Mivel mindent a

flow-t definiáló XML fájlban írunk le, ott mindenképpen jelölnünk kell, melyik állapotból melyikbe szeretnénk átmenni. Az esemény, ami kiválthatja a navigációt, a lekérésben érkezhet az `_eventId` paraméter segítségével (az MVC használatakor), vagy a JSF 1.2 használata esetén az ott leírt események érvényesek itt is: azaz vagy a gombok váltják ki őket, vagy a gombokban megadott metódusok visszatérési értékei kerülnek felhasználásra. Miután kiderült, melyik esemény következett be a nézeten, az SWF a definiált átmenetek (`tansition`) közül választ. A gombok által hívott metódusokat érdemes hanyagolni, helyette az ott elvégzendő üzleti logikát inkább a megfelelő `transition` elemen belül érdemes végrehajtani az `evaluate` vagy ritkább esetben a `set` elemek segítségével. Az így keletkező navigációs logika és a navigáció mellett elvégzendő üzleti folyamatok átláthatóbbak lesznek, mint az alap JSF esetén.

Feltételes navigációt az SWF esetében is kiválthatunk. Erre egy triviális megoldás a korábban már leírt `decision-state` állapot, amire sokkal inkább lehet azt mondani, hogy feltételes navigáció, ugyanis a JSF 2.0 esetén csak egy feltételt kötöttünk ki, hogy az adott navigáció számításba legyen-e véve a lehetséges navigációs szabályok között. Ezzel ellentétben az SWF-ben tényleges feltételes navigáció történhet, azaz a feltétel kiértékelése után két irányba lehet tovább haladni az állapotok között, de akár a többágú elágazásnak megfelelő szerkezetet is létrehozhatunk. Az SWF-ben is megadható viszont a JSF 2.0 feltételes navigációjával analóg működés. Ezt úgy lehet elérni, hogy a bekövetkezett felhasználói eseményhez tartozó megfelelő navigációs esetben, a valamelyik megadott `evaluate` elem által végrehajtott metódus visszatérési értékét a hamisnak megfelelő flow-eseményeket kiváltó értékekre állítjuk be. Minden esetben a visszatérési érték létrehoz egy eseményt, ami a `transition` elem végrehajtásában fog szerepet játszani. A különböző metódusok visszatérési értékei a 6.3. táblázat szerint konvertálódnak eseményekké. Ez a viselkedés egyébként azért van így, mert az SWF többek között az `evaluate` elemet is az Action interfészt implementáló osztály objektumaként képezi le.

A metódus visszatérési értékének típusa	Keletkező esemény
java.lang.String	a String értéke
java.lang.Boolean	yes true esetén, és no false esetén
java.lang.Enum	az Enum neve
bármely más típus	success

6.3. táblázat

Az érthetőség kedvéért erre leírok egy példát. Tegyük fel, van a 6.4. kódrészleten látható állapotunk. Ekkor, ha a bekövetkezett felhasználói esemény a `someAction`, az annak megfelelő `transition` elemben található kiértékelő elemek fognak lefutni. Először az első `evalute` fut le, és keletkezik egy esemény, amit felhasznál a flow, azaz ha a `couldIProceed` metódus igaznak megfelelő eseménnyel tér vissza, a következő kiértékelő tag is lefut és a `nextView` állapotban folytatódik a végrehajtás. Viszont, ha a visszatérési érték a hamisnak megfelelő esemény, akkor a további `evalute` elemek végrehajtása felfüggesztésre kerül, és az aktuális nézet fog újrarenderelődni.

```
<view-state id="viewName" view="viewName.jsp">
  <transition on="someAction" to="nextView">
    <evaluate expression="service.couldIProceed()"/>
    <evaluate expression="service.method()" result="flowScope.result"/>
  </transition>
</view-state>
```

6.4. kódrészlet

6.2. Életciklus

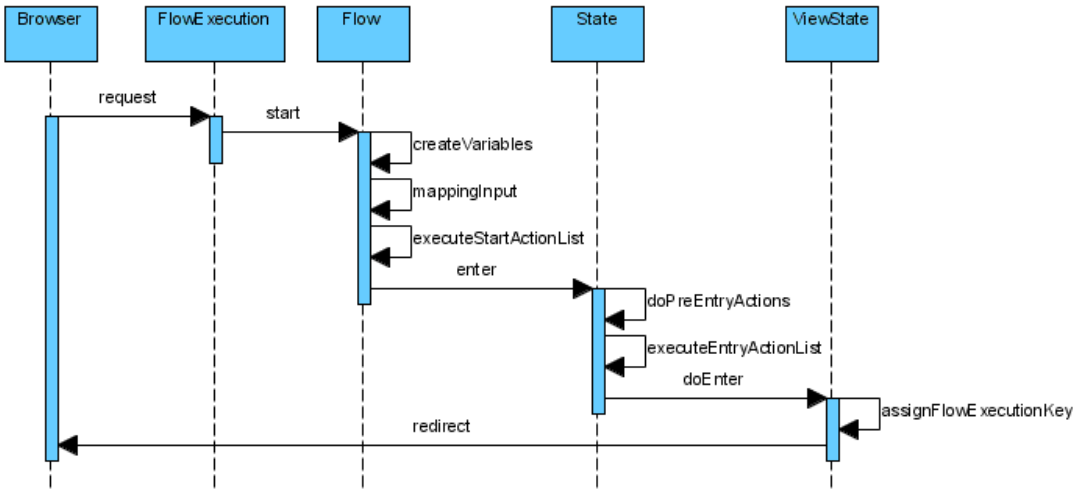
A JSF 2.0 életciklusa nem tér el az 1.2-ben megismerttől, csak pár új lehetőség lett hozzátéve. Ilyen lehetőség többek között az új események, a komponens fa részleges feldolgozása, és néhány más apróbb dolog, amiről még lesz szó később.

Sokkal érdekesebb az SWF életciklusa a JSF 1.2 használatával, azaz hogy a JSF életciklus mely részei mikor és hogyan futnak le. Alapvetően az SWF mint állapotgép működése magasabb szinten elég nyilvánvaló. Amennyiben a beérkező kérést a Dispatcher Servlet az SWF-hez rendelte hozzá, a korábban a konfigurációs résznél már leírt FlowExecution jön létre, vagy töltődik be a FlowExecutionRepository-ból. A FlowExecution implementációjának tartalmazni kell a FlowDefinition implementációnak egy példányát, ami leggyakrabban az XML flow-definíció reprezentációja, azaz a Flow osztály példánya.

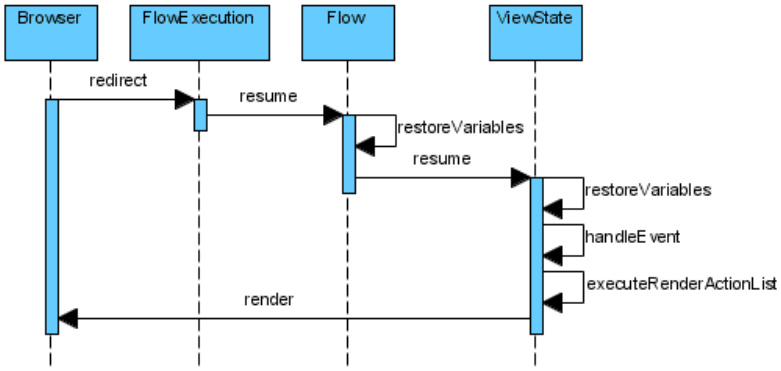
A flow első hívásakor létrejönnek a flow-változók, átadásra kerülnek az esetleges bemeneti paraméterek, végrehajtódnak az `on-start` részben megadott események, majd indításra kerül az első állapot, azaz az induló állapot.

Az állapoton belül lefutnak az elő belépési események, `view-state` esetén ezek a nézetváltozók, majd a belépési események, azaz az `on-entry` elemben megadott `evaluate` elemek. Végül a belépés is megtörténik. `View-state` esetén a nézet megjelenítésre és elküldésre kerül a kliensnek, amennyiben a `redirect` attribútum hamisra volt állítva, ellenkező esetben a kliens csak egy redirect üzenetet kap az új állapot kulcsával kiegészített URL-el (6.5. ábra). A nézet megjelenítésétől függ ugyanis, mikor kerül végrehajtásra az `on-render` tag (6.6. ábra)(redirect után, vagy annak hiányában az állapotba belépést követően).

A kliens miután megkapta a választ, valamilyen tetszőleges eseményt válthat ki, ami aztán egy új lekérést eredményez a szerver felé. A lekérés tartalmazza a flow aktuális állapotának egyedi azonosítóját valamint a kiváltó eseményt. Az azonosító segítségével a FlowExecutionRepository-ból betöltődik az előzőekben elmentett flow állapota, és folytatódik a korábban felfügesztett állapotban. Az eseménynek megfelelő átmenet bekövetkezik, a hozzá megadott kiértékelések elvégzésre kerülnek, majd vagy új állapotba kerülünk, ami előtt végrehajtódnak az `on-exit` elemben megadott kifejezések, vagy az aktuális nézet újra megjelenítésre kerül a `redirect` attribútumnak megfelelően azonnal, vagy csak redirect után. A redirect közben az esetleges felhasználói események a `flashScope`-ban tárolódnak (scopeokról később).

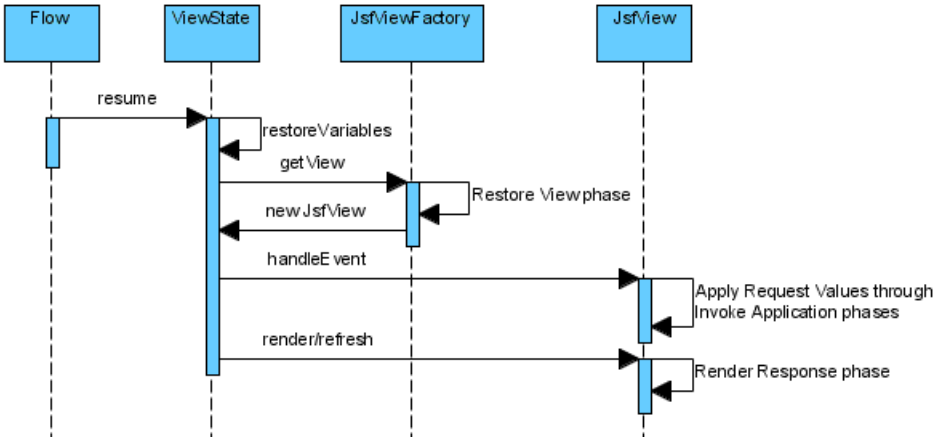


6.5. ábra. Nézet megjelenítési folyamata redirect előtt



6.6. ábra. Nézet megjelenítési folyamata redirect után.

Most már ismertettem a sima SWF életciklust, de felmerül a kérdés, ez hogy működik együtt a JSF 1.2 életciklusával? Az SWF a JSF életciklusát három nagy részre bontja szét, amik között a FacesContext nem létezik, és ebből következően nem is elérhető a flow belsejéből. Emlékezzünk vissza a Spring Faces beállításaira, ahol megadtuk, hogy a nézeteket gyártó osztályt a JsfViewFactoryCreator hozza létre. Ez minden `view-state` számára létrehoz egy JsfViewFactory osztályt, amiben tárolásra kerül az aktuálisan használt JSF életciklus, és a használni kívánt nézet neve is. A JsfViewFactory az életciklus és a nézet nevének felhasználásával legyártja a View interfészt implementáló JSF specifikus JsfView objektumot. Mikor a `view-state` állapot elindul vagy folytatódik, akkor történik meg a JSF életciklus első lépése, a Restore View fázis. A JsfView tárolja a nézethez tartozó UIViewRoot objektumot is, aminek segítségével a ViewState eseménykezelése közben lefut a 4 köztes fázis (esetleg a JSF életciklus ebben a részében kimaradhatnak fázisok, megfelelően a JSF beállításainak), azaz az Apply Request Values, a Process Validators, az Update Model Values és az Invoke Application. Szóval ebben a lépésben kerülnek végrehajtásra a JSF komponenseiben beállított validációk és események is. Végül az életciklus utolsó része, a Render Response fázis, a JsfView `render()` metódusa segítségével kerül végrehajtásra, ami a `view-state` állapotba való belépéskor (ha a redirect attribútum hamis) vagy annak a redirect utáni folytatásakor fut le (6.7. ábra).



6.7. ábra. JSF életciklus az SWF-ben

6.3. Scope-ok

A JSF 2.0 egy új scopeot vezet be, a meglévő szokásos három mellett (`application`, `session`, `request`). Ez a scope a `view` scope, ami tovább elérhető, mint a `request` scope, de nem él olyan soká, mint a `session` scope. Az itt elhelyezett adatok addig élnek, amíg a felhasználó be nem fejezi az adott nézeten való operálást. Programkódban a `view` scope elérése az `UIViewRoot.getViewMap()` metódus segítségével történhet, vagy implicit EL változó segítségével, amiről később.

A JSF 2.0-ban létezik egy scope szerűen használható tároló objektum, a `flash`. Az itt tárolt adatok csak a kliens sessionjében következő nézet számára lesznek elérhetőek. A következő nézet lehet ugyan az a nézet is, ebben az esetben az itt tárolt objektumok a redirectet élik túl. Az előző nézet objektumait tartalmazó `flash` a következő nézet renderelési fázisában törlődik. Programkódból az `ExternalContext.getFlash()` metódus segítségével érhető el.

JSF 2.0 biztosítja saját scope-ok létrehozását is. Annotációkkal ez a `@CustomScoped` segítségével állítható be, egészen pontosan a `value` attribútumában egy általunk kezelt Map típusú objektumot kell megadnunk value expression segítségével (6.8. kódrészlet). A megadott táblázat majd az objektumok tárolására fog szolgálni. Mikor az idő elérkezik egy ilyen objektum inicializására, a rendszer automatikusan létrehozza a beant és belerakja a táblázatba a megfelelő névvel, ha még nem létezett ilyen objektum a táblázatban.

```
@ManagedBean
@CustomScoped(value="#{customScope}")
public class TestBean {…}
```

6.8. kódrészlet

Az SWF-ben négy eddigiektől eltérő scope található meg az `application`, `session` és `request` scopeokon túl. Az első és legtágabb a `conversationScope`. Ez a scope a flowláncban első flow indulásakor jön létre és ugyanezen flow véget érésekor semmisül meg. Az összes alflow elérheti az itt tárolt objektumokat. A `session` scopeban

tárolódik, és az itt tárolt objektumoknak nem kötelező jelleggel, de érdemes kiterjesztenie a Serializable interfészt. A többi scopeot használó objektumnak viszont kötelezően ki kell terjesztenie a Serializable interfészt, ugyanis a lekérések között ezek az objektumok tárolódnak. A `conversationScope` után következik a `flowScope`, ami annál egy fokkal szűkebb, ez is a flow indulásától a flow végetéréséig tárolja az objektumokat, de csak az aktuális flown belül érhető el. Következő szűkebb scope a `viewScope`. Az itt tárolt objektumok csak a `view-state` állapoton belül elérhetőek. A `view-state` indulásától kezdve ugyanezen állapot végéig tárol. Ezeken kívül van egy további scope, a `flashScope`. Szintén a flow kezdetétől a végéig létezik, de minden egyes nézet renderelése után törlődik. Itt a hibaüzeneteket érdemes tárolni, ugyanis nem kell később azzal foglalkoznunk, hogy mi került a scopeba előzőleg, azt a rendszer úgyis törli.

6.4. Konverterek

A konverterek felépítése mindkét technológia mentén azonos. Mindig a nézeten található adatokat konvertálják a modellben megtalálható objektumokra és vissza. Azaz mindig egy stringből állítják elő a megfelelő objektumot, vagy stringbe konvertálják át azt.

A JSF a konverzióra a `javax.faces.convert.Converter` interfészt nyújtja. Ezeket a konvertereket minden egyes UI komponens számára külön megadhatjuk. A konverziók a Process Validations fázis során hajtódnak végre, miután minden komponens kinyerte a számára fontos adatokat a lekérésből (ez mind az 1.2.-es mind a 2.0.-ás verziókban így működik). Konverziós hiba esetén, ami a ConverterException kivétel bekövetkeztekor történik, a Render Response fázisba lép a végrehajtás, ami újra megjeleníti a nézetet a bevitt adatokkal és esetleges hibaüzenetekkel.

Saját konvertert több féle képen is meg lehet adni. Egyik lehetőség a faces-config.xml konfigurációs fájlban beállítani egy azonosítót a konverternek (6.9. kódrészlet), amit aztán majd a nézeten az UI komponensek `converter` attribútumában (6.10. kódrészlet), vagy a komponensen belül megadott `f:converter` elem `converterId` attribútumában adhatunk meg (6.11. kódrészlet).

```
<converter>
  <converter-id>myConverter</converter-id>
  <converter-class>jsf.utils.MyCustomConverter</converter-class>
</converter>
```

6.9. kódrészlet

```
<h:inputText value="#{backingBean.property}" converter="myConverter"/>
```

6.10. kódrészlet

```
<h:inputText value="#{backingBean.property}">
  <f:converter converterId="myConverter"/>
</h:inputText>
```

6.11. kódrészlet

Az UI komponens `converter` attribútumában value expression segítségével is megadható a konverter, ez akkor lehet jó, ha a háttér beanben létrehozunk a konverterünkből egy tulajdonságot, például egy névtelen osztály implementálásával.

Ha a konverter, amit írtunk, olyan jól sikerült, akkor azt beállíthatjuk alapértelmezett konverternek is (6.12. kódrészlet). Az alapértelmezett konverterek futnak le mindig, mikor nem állítottunk be mást a komponenseknél. Konverzió során a JSF felismeri, milyen osztály objektumára kell konvertálni a lekérésben beérkezett adatot, és az osztály alapján keres beállított konvertereket a konfigurációban.

```
<converter>
  <converter-for-class>
    model.ClassThatsNeedsKonversion
  </converter-for-class>
  <converter-class>jsf.utils.MyCustomConverter</converter-class>
</converter>
```

6.12. kódrészlet

A JSF több beépített konvertert is tartalmaz. Az összes primitív típushoz létezik konverter, és a BigDecimal és BigInteger osztályokhoz is. Az egyik legfontosabb konverter viszont a dátumok konvertálására alkalmas `f:convertDateTime`, aminek `pattern` attribútumával állíthatjuk be a megjeleníteni kíván dátum kinézetét. A másik lényeges konverter a `f:convertNumber`, ami segítségével tagolhatjuk, vagy valuta kinézetre formálhatjuk a számokat.

A JSF 2.0.-ban a konverter osztályainkat nem csak XML-ben állíthatjuk be, hanem annotáció segítségével is. Erre a `@FacesConverter` való, aminek `value` attribútuma a konverter azonosítójának felel meg, a `forClass` attribútumában pedig a konvertálandó osztály nevét adhatjuk meg.

Az SWF-ben a konverzió előtt a modell és a nézet közti kapcsolat kiépítésére van szükség. Ez a `view-state model` attribútumával történhet, segítségével egy elérhető scopeban lévő beant rendelhetünk hozzá a nézethez, és az űrlapokon található adatok validációját és konverzióját hozzáköthetjük a modell elem metaadataihoz. Ilyenkor a kötések implicit történnek, explicit kötést a `binder` alelem használatával válthatunk ki (6.13. kódrészlet). A kötés a következő folyamatot indítja el a nézeten történt esemény kiváltódása után: Először a beérkezett lekérés adatai hozzárendelésre kerülnek a modell attribútumaihoz, ekkor történik a konverzió, majd a konverzió után a validáció következik be. Konverziós hiba esetén a nézet újra megjelenítésre kerül.

```
<view-state id="viewId" view="viewname.xhtml" model="scopedBean">
  <binder>
    <binding property="beanProp1" converter="MyConverterId"/>
    <binding property="beanProp2" converter="AnotherConverter"/>
  </binder>
</view-state>
```

6.13. kódrészlet

Az SWF is tartalmazza a primitív típusokhoz használható konvertereket, továbbá a BigInteger, BigDecimal, Enum és Date konvertálásához szükségeseket is. A Spring Faces használata esetén még egy plusz konvertert is alkalmazhatunk, a DataModel konvertálására. Saját konverter alkalmazása viszont már nem olyan egyszerű, mint a JSF esetében. A konvertereknek az `org.springframework.binding.convert.converters.TwoWayConverter` interfészt kell kiterjeszteniük.

A saját konvertert megadni a ConversationService interfészt kiterjesztve lehet, amit majd aztán a `flow-builder-services` beállításnál kell megadni (6.15. kódrészlet). Ennek legegyszerűbb megvalósítása a DefaultConversationService osztály kiterjesztése az `addDefaultConverters()` metódus felülírásával (6.14. kódrészlet). Minden egyes konverter külön azonosítóval rendelkezik, amivel majd hivatkozhatjuk. Ha explicit nem adunk meg konvertert, akkor az osztály alapján dől el, melyik konverter kerül használatra. A konverterek azonosítója az osztály nevéből származik. Minden konverterhez megadható álnév is, hogy könnyebb legyen hivatkozni (pl.:java.util.Date-hez a date).

```
public class MyConversationServiceClass extends DefaultConversionService{
      @Override
      protected void addDefaultConverters() {
            super.addDefaultConverters();
            addConverter(new MyConverter())
      }
}
```

6.14. kódrészlet

```
<faces:flow-builder-services id="facesFlowBuilderServices"
    conversion-service="myConversationService"/>
<bean name="myConversationService"
    class="package.MyConversationServiceClass"/>
```

6.15. kódrészlet

A flown belül a kiértékelések visszatérési értékét is konvertálhatjuk. Itt egyszerűn csak a `result-type` attribútum megadására van szükség az `evaluate` elemben (6.16. kódrészlet).

```
<evaluate expression="serviceBean.listTableElements()"
    result="viewScope.table" result-type="dataModel"></evaluate>
```

6.16. kódrészlet

A `transition` elemben, ha használjuk a `model` attribútumot addig nem haladhatunk tovább az állapotok között, amíg a konverzió nem hajtódik végre sikeresen. Ezt a működést felülbírálhatjuk minden egyes átmenet esetén a `bind` attribútummal. Ekkor az adott átmenet előtt nem fog konverzió történni.

```
<transition on="select" to="editElement" bind="false">
```

6.17. kódrészlet

6.5. Validáció

A JSF-ben a validáció hasonlóan működik, mint a konverterek, azaz a mag elemeinek segítségével. Az 1.2.-ben három validátor csatolható a komponensekhez: a `f:validateDoubleRange`, a `f:validateLongRange` és a `f:validateLength`. Ezek sorban double típusú számok, long típusú számok, és a kliens által begépelt szöveg hosszának (6.18. kódrészlet) az ellenőrzésére valók. A 2.0. ezt további három validátorral bővíti ki, ezek a `f:validateRequired`, a `f:validateRegexp` és a `f:validateBean`. Az első azt ellenőrzi kivan-e töltve a mező, a második pedig reguláris kifejezések segítségével ellenőrzi a hozzá tartozó mezőt.

```
<h:inputText value="#{bean.property}">
  <f:validateLength maximum="5" minimum="10"/>
</h:inputText>
```

6.18. kódrészlet

A felhasználónak, hogy egy komponenst kötelező jelleggel ki kelljen tölteni, nem szükséges külön validátor elemet megadni, elég csak a komponens `required` attribútumát igazra állítani. Az összes JSF tag támogatja ezt az attribútumot, továbbá a validátorokkal is használható kombinálva.

A `f:validateBean` elem az úgynevezett Bean Validation JSR (JSR-303) használatának finomhangolására vezették be, ami egy generikus, rétegfüggetlen mechanizmus adatok validálására. A szabvány magába foglal több annotációt is (mint pl.:`@Size`, `@NotNull`, `@Min`, `@Max` stb.) A JSF 2.0. ezt a szabványt beépített módon támogatja, azaz ha a JSR-303 implementáció jelen van a könyvtárszerkezetben a JSF automatikusan validálja az UIInput komponensek által hivatkozott és szabvány szerint annotált értékeket. Szóval a `f:validateBean` elem `validateGroups` attribútuma által finomhangolhatjuk, melyik validációs csoport legyen használatban (További információk a validációs csoportokból a JSR-303 API-ban). Mivel a JSR-303 használatával validálható null és üres érték is, ezért az implementáció jelenlétekor a JSF is engedélyezi az ilyen értékek validálását. Ez problémákat okozhat régi validátorok használata esetén, ezért ez a viselkedés kikapcsolható a `javax.faces.VALIDATE_EMPTY_FIELDS` környezeti változó segítségével.

Saját validátort a `javax.faces.validator.Validator` interfész kiterjesztésével, majd faces-context.xml-ben regisztrálásával készíthetünk (6.19. kódrészlet), hasonlóan a konverterekhez.

```
<validator>
  <validator-id>myValidator</validator-id>
  <validator-class>package.MyValidatorClass</validator-class>
</validator>
```

6.19. kódrészlet

A validátorban validációs hibát a ValidatorException kivétellel válthatunk ki, ekkor, mint a konverterek esetében, az aktuális nézet újra megjelenítésre kerül a hibaüzenetekkel. A saját validátort a konfigurációs beállítás után a `f:validator` elem segítségével használhatjuk (6.20. kódrészlet).

```
<h:inputText value="#{bean.property}">
  <f:validator validatorId="myValidator"/>
</h:inputText>
```

6.20. kódrészlet

Validáció elvégezhető a beanek metódusainak segítségével is, azaz nem szükséges validációs osztályt létrehozni (6.21. kódrészlet). A metódus neve tetszőleges lehet, de a szignatúrájának egyeznie kell a Validator interfészben lévő `validate()` metóduséval.

```
<h:inputText value="#{bean.property}"
    validator="#{bean.validatorMethodName}"/>
```

6.21. kódrészlet

JSF-ben a validáció elhagyására az `immediate` attribútum szolgál az eseményeket kiváltó UI komponensekben. Ekkor az adott esemény nem a megszokott Invoke Application fázisban fog lefutni, hanem korábban, az Apply Request Values fázisban.

Az SWF-ben is két validációs lehetőséget ad a keretrendszer. Metódus szintűt és osztály szintűt. A validáció is mint a konverzió a `model` megadása után történhet a modellben megadott osztályra. Se az osztály se a metódus írásához nem kell semmilyen interfészt írni, csak bizonyos elnevezési konvenciónak megfelelni.

Külön validációs osztály írásakor az elnevezési konvenciók a következők: Az osztály nevének a `view-state`-ben megadott modell nevével kell kezdődnie (nagybetűvel), kiegészítve a Validator szóval, azaz ${modelnév}Validator. A validátor osztályban aztán minden egyes állapothoz külön kell írni validációs metódust a validate${állapot} formában (6.22. kódrészletben látható állapothoz a 6.23. validátor tartozik). Majd az így megírt osztályt deklarálni kell, mint beant. Validáció során az SWF az ApplicationContext-ben található beanek nevei közül az aktuális modell nevét felhasználva választja ki a validátort, majd az állapot nevét felhasználva a megfelelő metódus hívását végzi el. A validátor osztályban tetszőleges számú metódus lehet. A metódusok a validálni kívánt objektumot és egy validációs környezetet kapnak meg. A validációs környezetbe helyezhetjük el hiba esetén az üzeneteket, valamint onnan nyerhetjük ki például a jelenlegi felhasználó adatait.

```
<view-state id="viewStateName" view="view.xhtml" model="modelName">
</view-state>
```

6.22. kódrészlet

Validációs metódust a validálni kívánt modellobjektumba kell írni. Hasonlóan az osztálybeli elnevezési konvenciókhoz itt is a validate${állapot} formában kell definiálnunk azt.

```
@Component
public class ModelNameValidator {
  public void validateViewStateName(Object modelObject,
      ValidationContext context){
  }
}
```
6.23. kódrészlet

A validáció elnyomható az egyes átmenetek beállításainál, így például ha a felhasználó a mégse gombra nyom, nem kerülnek validálásra az adatok. Az ilyen működés a `validatie` attribútum segítségével váltható ki (6.24. kódrészlet).

```
<transition on="cancel" to="otherState" validate="false"/>
```
6.24. kódrészlet

Éppenséggel a kiértékelő utasítások nem konkrét validációs eszközök, de használhatóak úgy, mintha validálnának. Mivel a kiértékelés mindig egy eseménnyel tér vissza, amit a flow motorja felhasznál, ezért ha a kiértékelésben olyan metódust hívunk meg, ami a flow állapotának működését leállítja, azt validációként használhatjuk. Az ilyen események bekövetkeztekor (pl.:`no` vagy `error`), amik leállítják az állapot további működését, az aktuális nézet újra fog renderelődni.

Az SWF Faces része tartalmaz kliens oldali validátorokat is. Ilyen validátor a bemeneti mezőkre vonatkozóan a `sf:clientTextValidator`, a `sf:clientDateValidator` (amire volt példa a 5.22. ábra), a `sf:clientNumberValidator` és a `sf:clientCurrencyValidator`, végül az összes kliens oldali validációs eszközt elsütő `sf:validateAllOnClick`, amit gombokon és linkeken használhatunk. Ezek a validátorok egyébként szintén reguláris kifejezések segítségével működnek.

6.6. Hibaüzenetek kezelése

A validáció és a konverzió során keletkezett hibákat valahogy érthetővé kell tenni a felhasználók számára a kliens oldalon. Erre JSF-ben a FacesMessage osztály való. Ez legtöbbször egy UI komponenssel áll kapcsolatban. A generált üzenetek egy sorban vannak nyilvántartva a FacesContext-ben, és a JSF életciklus végén megjelenítésre kerülnek a nézeten. A hibaüzeneteknek négy súlyossági foka van: information, warning, error, fatal. Továbbá minden üzenet tartalmaz összegzést, aminek egy gyűjtőfogalmat érdemes adni, és részleteket, amibe a hiba részletes okát kell leírni. (6.25. kódrészlet)

```
new FacesMessage(FacesMessage.SEVERITY_ERROR, "Validation error",
    "Value is required!");
```
6.25. kódrészlet

A létrehozott üzeneteket a nézeten a `h:message` és `h:messages` elemekkel jeleníthetjük meg. Az első esetben csak az utolsó egy üzenetet, a másodikban a FacesContext-

ben található összes üzenetet megjeleníthetjük, ami a JSF életciklus alatt keletkezett. Mindkét elemhez megadható a komponens, amihez tartozó üzenet(ek)et jelenítsen meg, valamint különböző beállítások az üzenet megjelenítésével kapocslatban (pl.: minden súlyosságú üzenet jelenjen meg, az üzenet leírása jelenjen meg csak stb.).

SWF-ben a flow végrehajtása közben keletkezett üzeneteket a MessageContext tárolja. Mind nemzetközisített, mind sima szöveges üzeneteket elhelyezhetünk itt. Az SWF MVC szerkezete miatt, az üzenetek megjelenítését teljes egészében a nézetre hagyja. Ellenben az Üzenetek készítésére létrehoztak egy a Builder tervezési mintát megvalósító üzenetépítő osztályt, a MessageBuilder-t. Segítségével könnyen összeállíthatunk üzeneteket. Megadhatjuk vele az üzenet forrását, súlyosságát, alapértelmezett szövegét és a szöveghez tartozó esetleges paramétereket is (6.26. kódrészlet).

```
new MessageBuilder().error().source("componentId")
    .defaultText("Value is required!").build();
```

6.26. kódrészlet

A Spring Faces használata esetén a MessageContext üzeneteit alapból nem tudnánk megjeleníteni a JSF oldalakon, ezért az SF-ben létrehozták a FlowFacesContextMessageDelegate nevezetű osztályt, amely a JSF és SWF üzenetek közti konverziót látja el, így nyugodtan használható az SWF üzenetkezelő rendszere.

6.7. Message Bundles

A nemzetköziesítés használatához érdemes kigyűjteni egy helyre, egy külső fájlba, a nézeten található feliratok szövegét. Ez a módszer segít a feliratok konzisztenciájának megőrzésében, valamint az esetleges más nyelvekre való lefordításban is. A szövegeket az osztályok hierarchiájában tetszőleges csomagba el lehet helyezni a .properties kiterjesztéssel.

A JSF 2.0.-ban nem vezettek be újabb eszközt a szövegek nemzetköziesítésére, így az 1.2.-ben már létezőt írom le, ami két lehetőség biztosított az I18N feliratok kezelésére. Az egyik a konfigurációs fájlokban a `<resource-bundle>` tag megadása a szöveg csomagjának nevével, és a később hivatkozható változó nevével (6.27. kódrészlet). Ennek alternatívája a nézeten megadható `<f:loadBundle>` elem, ahol az előbbi esethez megegyező információkat adhatunk meg (6.28. kódrészlet). A változó nevét aztán később a különböző komponensekben hivatkozva felhasználhatjuk.

```
<application>
  <resource-bundle>
    <base-name>package.messagesPropertiesFileName</base-name>
    <var>msgs</var>
  </resource-bundle>
</application>
```

6.27. kódrészlet

```
<f:loadBundle basename="package.messagesPropertiesFileName" var="msgs"/>
```

6.28. kódrészlet

A lokális nyelvi beállításokat tartalmazó szöveges fájlt megadni az ISO-639 két betűs nyelvi kód használatával lehet a fájl nevének végén egy _ karakterrel elválasztva, de a kiterjesztés neve előtt. Ezt a lokalizált fájlt aztán a faces-config.xml fájlban adhatjuk meg, a támogatott lokális beállítások segítségével (6.29. kódrészlet). Az így definiált lokalizált fájlok között aztán a JSF a http lekérések fejlécében található `Accept-Language` értéke szerint fog váltani. Explicit mi is megadhatjuk a használt lokalizált fájlt a `<f:view> locale` attribútuma segítségével.

```
<locale-config>
  <default-locale>hu</default-locale>
  <supported-locale>en</supported-locale>
</locale-config>
```

6.29. kódrészlet

Az SWF ezzel szemben automatikusan kezeli a lokalizációt. Ha a flow definíciós fájljával egy könyvtárban megadunk egy messages.properties fájlt és ennek további tetszőleges nyelvi változatát, akkor az SWF ezt automatikusan felismeri és betölti. A különböző nézeteken, vagy a flowban is a `resourceBundle` implicit EL változós egítségével fogjuk tudni hivatkozni a fájlban definiált kulcsokat.

Ha valamilyen oknál fogva nem akarjuk használni az SWF alapértelemzett lokalizációkezelését, akkor a Spring mag szolgáltatásaihoz kell fordulnunk, azaz az ApplicationContext-éhez. Egy tetszőleges névvel ellátott beant kell definiálnunk az org.springframework.context.support.ResourceBundleMessageSource osztállyal. Ennek az osztálynak van egy `basenames` tulajdonsága, amiben megadhatjuk a használni kívánt nyelvi fájlok neveit (6.30. kódrészlet). Az így definiált beant és a lokalizációs fájlokban lévő kulcsokat utána már hivatkozhatjuk a megfelelő módon.

```
<bean id="myMsgBundle"
  class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>package.messagesPropertiesFileName</value>
      <value>package.secondMessagesPropertiesFileName</value>
    </list>
  </property>
</bean>
```

6.30. kódrészlet

6.8. Resource kezelés

A JSF 1.2.-ben még nem lehetett szabványosan kezelni az oldalak megjelenéséhez szükséges különböző képeket, fájlokat azaz resourceokat. A 2.0.-ban viszont már bevezettek pár eszközt ezek kezelésére. A resourceok kezelésére hozták létre a ResourceHandler API-t, ami Resource objektumokat kezel, amik pedig az egyes resourceokat reprezentálják, azaz a scripteket és stílusokat. Ez az API kerül meghívásra az új resourcokat kezelő elemekben is. Az összes resource kérése a FacesServleten keresztül történik, így használatával többet nincs

szükség külön filterek, servletek vagy fázis eseményfigyelők írására, valamint a szükséges fájlok egybe is csomagolhatóak a webalkalmazással.

Az alapértelmezett implementáció szerint a kért resourceoknak az alkalmazás gyökerében a resources, csomagolt alkalmazás esetén a META-INF/resources könyvtáron belül kell lenniük. A resource azonosításához a fájloknak a [localePrefix/] [libraryName/] [libraryVersion/] resourceName [/resourceVersion] mintának kell megfelelniük. Mint látható az egyetlen kötelező rész a resource neve.

A resourceokat a nézeten a `<h:outputScript>` és a `<h:outputStylesheet>` segítségével helyezhetjük el, az előbbivel scripteket az utóbbival stílusokat. Az outputScript esetén megadhatjuk a script kívánt betöltési helyét is a `target` attribútummal (pl.: `target="head"` esetén a html fej részében kerül renderelésre az elem tartalma). Továbbá a komponens maga is elhelyezhet resourceokat az oldalon, ugyanis a komponensek fejlesztői a `@ResourceDependency` annotációval a komponens osztályában megadhatják a szükségese script fájlokat a komponens működéséhez. Ugyan azt a resourcet az előzőekből kifolyólag több komponens is csatolhatja, viszont a rendszer ezeket észleli, és a többször csatolt elemeket kiszűri.

Az SWF resource kezeléséről már volt szó a „JSF integráció - Spring Faces” fejezetben így itt csak összefoglalom az ott leírtakat. Konkrétan az SWF nem kezel resourceokat, ezt a Spring JS és a Spring Faces csomagok végzik. Az összes resource kérés, amit a Spring Faces JSF kiegészítés komponensei generálnak a ResourceServlet-hez érkezik, azaz ez a servlet szolgálja ki azokat. A resource fájloknak itt nem kell fixen meghatározott mappában lennie, tetszőleges elérési út alkalmazható, és tömörítve is lehetnek a fájlok. A resourceok elérésére az `includeStyles` és az `includeScripts` komponensek valók, ajánlás szerint az oldal fej részében alkalmazva.

6.9. EL Objektumok

Mindkét rendszerben találhatóak implicit EL objektumok. Ezek az objektumok arra szolgálnak, hogy elérhetővé tegyenek bizonyos szolgáltatásokat value expressionök segítsségével a nézeteken vagy a navigációban. A JSF 2.0.-ban ezek az objektumok a következőek (az ImplicitObjectELResolver osztály definiálja őket, kivéve a flasht, azt a FlashELResolver oldja fel, a 6.31. táblázat tartalmazza, melyik objektum, melyik scopeban található, valamint milyen java kóddal érhetőek el a beanekből):

- `application`: Az alkalmazás környezetét tartalmazó objektum.
- `applicationScope`: Az `application` scopeot reprezentáló táblázat.
- `cc`: A jelenleg feldolgozás alatt álló komponensnek a legközelebbi ős kompozit komponense.
- `component`: A jelenleg feldolgozás alatt álló komponens.
- `cookie`: A jelenlegi lekéréshez csatolt sütik neveinek nem módosítható táblázata.
- `facesContext`: A feldolgozás alatt álló lekéréshez tartozó összes információ.

- `flash`: Az alkalmazáshoz tartozó `flash` objektum.
- `header`: A lekéréshez tartozó fejlécek nem módosítható táblázata, a fejléc kulcsához csak az első értéket tartalmazza.
- `headerValues`: A lekéréshez tartozó fejlécek nem módosítható táblázata, a fejléc kulcsához tartozó összes értéket tartalmazza.
- `initParam`: Az alkalmazás inicializáló paramétereinek (web.xml) nem módosítható táblázata.
- `param`: A lekérés paramétereinek nem módosítható táblázata, a paraméter kulcsához csak az első értéket tartalmazza.
- `paramValues`: A lekérés paramétereinek nem módosítható táblázata, a paraméter kulcsához tartozó összes értéket tartalmazza.
- `request`: A lekérésből készült környezetfüggő objektum.
- `requestScope`: A `request` scopeot reprezentáló táblázat.
- `resource`: A resourceokat kezelő ResourceHandler singleton objektum.
- `session`: A jelenlegi lekéréshez tartozó `session` objektum.
- `sessionScope`: A `session` scopeot reprezentáló táblázat.
- `view`: A jelenlegi nézet komponensfájának gyökere.
- `viewScope`: A `view` scopeot reprezentáló táblázat.

Objektum	Scope	Ekvivalens kód
application	Application	externalContext.getContext()
applicationScope	Application	externalContext.getApplicationMap()
cc	?	UIComponent.getCurrentCompositeComponent(facesContext)
component	?	UIComponent.getCurrentComponent(facesContext)
cookie	Request	externalContext.getRequestCookieMap()
facesContext	Request	FacesContext.getCurrentInstance()
flash	?	externalContext.getFlash()
header	Request	externalContext.getRequestHeaderMap()
headerValues	Request	externalContext.getRequestHeaderValuesMap()
initParam	Application	externalContext.getInitParameterMap()
param	Request	externalContext.getRequestParameterMap()
paramValues	Request	externalContext.getRequestParameterValuesMap()
request	Request	externalContext.getRequest()
requestScope	Request	externalContext.getRequestMap()
resource	?	facesContext.getApplication().getResourceHandler()
session	Session	externalContext.getSession(true)
sessionScope	Session	externalContext.getSessionMap()
view	Request	facesContext.getViewRoot()
viewScope	?	facesContext.getViewRoot().getViewMap()

6.31. táblázat

A Spring Web Flowban elérhető EL objektumok, feloldásukért az ImplicitFlowVariableELResolver a felelős, kivéve a `flowRequestContext`-et amiért a

RequestContextELResolver és a `resourceBundle`-t amiért pedig a FlowResourceELResolver osztály a felelős:

- `conversationScope`: A `conversationScope`-ot reprezentáló táblázat.
- `currentEvent`: A jelenleg érvényes esemény, amit a felhasználó váltott ki.
- `currentUser`: A bejelentkezett felhasználó adatait tartalmazó objektum.
- `externalContext`: Az SWF-et meghívó környezetet érhetjük el vele.
- `flashScope`: A `flashScope`-ot reprezentáló táblázat.
- `flowExecutionContext`: A flow jelenlegi állapotát reprezentáló objektum eléréséhez használható.
- `flowExecutionUrl`: Az aktuális `view-state`-hez tartozó környezet relatív URI.
- `flowRequestContext`: Az SWF futásának információit tartalmazó objektum.
- `flowScope`: A `flowScope`-ot reprezentáló táblázat.
- `messageContext`: Üzenetek eléréséhez szolgál.
- `requestParameters`: A kliens által küldött lekérést tartalmazó objektum.
- `requestScope`: A `request` scopeot reprezentáló táblázat.
- `resourceBundle`: Az üzenetekhez tartozó resourceokat érhetjük el vele (I18N).
- `viewScope`: A `viewScope`-ot reprezentáló táblázat.

Objektum	Ekvivalens kód
conversationScope	requestContext.getConversationScope()
currentEvent	requestContext.getCurrentEvent()
currentUser	externalContext.getCurrentUser()
externalContext	requestContext.getExternalContext()
flashScope	requestContext.getFlashScope()
flowExecutionContext	requestContext.getFlowExecutionContext()
flowExecutionUrl	requestContext.getFlowExecutionUrl()
flowRequestContext	RequestContextHolder.getRequestContext()
flowScope	requestContext.getFlowScope()
messageContext	requestContext.getMessageContext()
requestParameters	requestContext.getRequestParameters()
requestScope	requestContext.getRequestScope()
resourceBundle	requestContext().getActiveFlow().getApplicationContext()
viewScope	requestContext.getViewScope()

6.32. táblázat

6.10. Ajax

A JSF 1.2.-ben nem volt még szabványos AJAX kezelés. Ez ahhoz vezetett, hogy a kívülálló JSF komponensfejlesztők mind saját megoldást nyújtottak a problémára, ami aztán kompatibilitási nehézségekbe torkollott. Miután szinte minden komponenskönyvtárank különböző AJAX-ot támogató rendszere van, két különböző komponenskönyvtárat nehéz

együtt használni. Erre a kaotikus állapotra nyújt megoldást a JSF 2.0. ahol már a szabvány is tartalmaz AJAX-os megoldást. A JSF 2.0. ugyanis tartalmaz kliens oldali JavaScript API-kat is. AJAX kérés kiváltásához a `jsf.ajax.request()` JS függvény lett definiálva, ami összegyűjti az összes szükséges adatot a kliens oldalon, és azokat elküldi a szervernek. A szervel oldalon az AJAX kérést a PartialViewContext osztály fogja kezelni. A kérésből készült PartialViewContext objektum tartalmazza, mely komponenseket kell újra feldolgozni és renderelni. A részleges feldolgozást az előbb nevezett osztály `processPartial()` metódusa indítja el. Az elkészült választ a szerver visszaküldi a kliensnek, ami majd frissíti a változott DOM objektumokat.

A `jsf.ajax.request()` függvény hívása nehézkes lenne minden alkalommal, ezért a szabvány tartalmaz egy `<f:ajax>` elemet is. Segítségével egy vagy több UI komponenshez köthetünk AJAX viselkedést. A `render` attribútumban a frissíteni kívánt komponensek ID-ját adhatjuk meg (használható kulcsszavak a `@this`, `@form`, `@all`, `@none`, az aktuális, az űrlapon lévő összes, az oldalon lévő összes, semelyik komponens frissítésére), az `event` attribútumba pedig az esemény által kiváltandó akciót, amit majd a navigáció felhasználhat. (6.33. kódrészlet)

```
<h:inputText value="#{bean.property}">
  <f:ajax execute="@form" render="myform:message" event="click"></f:ajax>
</h:inputText>
<h:outputText id="message" value="#{bean.ajaxMessage}"></h:outputText>
```

6.33. kódrészlet

A SWF kliens oldali AJAX kérés indítására a Spring JS modult használja, ami három js script fájlt használ (a script fájlok URI-je a 6.34. kódrészleten látható). Hogy a komponenshez hozzárendeljünk AJAX kérést a Spring csomagot kell használnunk, azon belül is az `AjaxEventDecoration()` függvényt. Az `elementId` a kiváltó komponens azonosítója, az `event`, hogy melyik eseményre lépjen működésbe az AJAX lekérés, és a `params`-on belül a `fragments`, hogy melyik elemek frissüljenek (6.35. kódrészlet).

```
/resources/dojo/dojo.js
/resources/spring/Spring.js
/resources/spring/Spring-Dojo.js
```

6.34. kódrészlet

```
<a id="someLink" href="spring/somelink">SomeLink</a>
<script type="text/javascript">
  Spring.addDecoration(new Spring.AjaxEventDecoration({
    elementId: "someLink",
    event: "onclick",
    params: { fragments: "body" }
  }));
</script>
```

6.35. kódrészlet

A Spring Faces használatakor az előző bekezdésben leírt AJAX lekérést kiváltó függvények sokkal könnyebben megadhatóak, ugyanis a SF komponensek alapból ezt a html kódot generáljhatják. A Spring Facesről már írtam a „JSF integráció - Spring Faces” fejezetben. Viszont amiről nem írtam az, hogy a szerver oldalon mi történik egy AJAX kérés bekövetkezésekor. A lekérések beérkezésekor az SWF az AjaxHandler nevezetű interfészt használja arra, hogy eldöntse az aktuális lekérés AJAX kérés-e vagy sem. Mivel sok komponens könyvtár különböző módon jelzi, hogy az aktuális kérés AJAX kérés, a komponens könyvtárak SWF integrációjához ezt az interfész kell kiterjeszteni, az aktuális komponenskönyvtárnak megfelelő módon (például a Trinidad komponenskönyvtár integrációjához a 6.36. kódrészlet szerint kell megírni az AjaxHandler-t).

```
public class ApacheTrinidadAjaxHandler extends SpringJavascriptAjaxHandler{
  public boolean isAjaxRequest(HttpServletRequest request,
      HttpServletResponse response) {
    String trXHRMessageHeader = request.getHeader("Tr-XHR-Message");
    if (trXHRMessageHeader!=null && trXHRMessageHeader.equals("true")) {
      return true;
    } else {
      return super.isAjaxRequest(request, response);;
    }
  }
}
```

6.36. kódrészlet

Miután az SWF az AjaxHandler segítségével kiderítette, hogy az aktuális kérés AJAX kérés, a `view-state` állapotokban megadott `render` alelem használatával, a nézet megjelenítési technikától függően, csak az adott nézet egy részét fogja frissíteni. A frissítés eredményét az SWF elküldi a kliensnek, ahol aztán a megfelelő DOM elemek felülírásra kerülnek.

6.11. Állapotmentés

Mind a JSF mind az SWF elmenti a nézethez tartozó összes objektumot. A JSF-ben két féle állapotmentési lehetőség van, a kliens oldali és a szerver oldali. A kettőt beállítani a `STATE_SAVING_METHOD` környezeti paraméter `client` vagy `server` értékekre állításával lehet (6.37. kódrészlet). A kliens oldali esetben a komponensek kódolásra kerülnek és egy rejtett komponensben lesznek tárolva a klienshez elküldött nézeten. Ez a módszer jelentősebben megterheli a hálózatot, viszont így a szervernek kevesebb memóriára lesz szüksége. A szerver oldali tárolás esetén a komponensek a memóriában tárolódnak el.

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>server</param-value>
</context-param>
```

6.37. kódrészlet

Az előzőekben leírt módszer nem volt túl jó, ugyanis az 1.2. verzióban még az összes komponens mentésre került, ami jelentős méreteket ölthetett, így kevéssé praktikussá téve a

kliens oldali állapotmentést, továbbá a komponensek `saveState` és `restoreState` metódusainak írása fölöttébb kellemetlen, és könnyen elrontható (StateHolder interfészt implementáló osztályokban, azaz az összes komponensben). A JSF 2.0. viszont megoldja ezt a problémát a részleges állapotmentés segítségével. Ennek lényege, hogy a komponens fa mindig visszaállítható a kezdő állapotába a nézet újraépítésével, így az összes olyan komponensállapot elmentése fölösleges, ami a kezdő állapotában van. Mivel legtöbb esetben a komponensek nagyobb része ritkán szokott változni, ezért csak az idáig változott komponensek állapotait menteni sokkal helytakarékosabb. A változás mentését a PartialStateHolder interfészt implementáló komponensek valósítják meg, amikben a `markInitialState()` metódus akkor kerül meghívásra, mikor a komponens a kezdő állapotba került. A másik változás a StateHelper interfész bevezetése, ami egy táblázatszerű objektum, ami a komponens állapotának tárolásában segít, és megszünteti a saját `saveState` és `restoreState` metódusok biztosításának szükségességét.

Az SWF az állapotok mentésére a FlowExecutionRepository interfészt használja. Ez az interfész a flow végrehajtásának állapotait (FlowExecution) szerializálja ki és állítja vissza. A flow minden egyes végrehajtási állapota a működő flowk egy aktuális állapotának reprezentációja. Az alapértelmezett implementációja az előbbi interfésznek a DefaultFlowExecutionRepository

A DefaultFlowExecutionRepository-t a beállításoknál ismertetett `flow-executor` elem segítségével lehet konfigurálni, azon belül is a `flow-execution-repository` segítségével. Ez az osztály az egy felhasználóhoz tartozó flowvégrehajtásokat a ConversationManager implementációval tartja számon, ahol beállítható (`max-executions` attribútum) maximálisan hány flow futhat egy felhasználónak. Itt kerül kiosztásra a FlowExecution objektumoknak a párbeszéd azonosító, ami majd a flow azonosítókulcsának a része lesz. További feladata az osztálynak, hogy a flow végrehajtásának állapotait perzisztálja, pillanatképek készítésével. A pillanatképek betöltésével utána folytatható a végrehajtás. Ez a perzisztáció oldja meg visza gomb problémáját is, mert ha a felhasználó a vissza gombra kattint az SWF csak visszatölti a megfelelő kulccsal rendelkező pillanatképet. A pillanatképek maximális számát is megadhatjuk a `max-execution-snapshots` attribútummal. (6.38. kódrészlet)

```
<webflow:flow-executor id="flowExecutor" flow-registry="flowRegistry">
  <webflow:flow-execution-repository max-execution-snapshots="10"
                                     max-executions="5"/>
</webflow:flow-executor>
```

6.38. kódrészlet

A DefaultFlowExecutionRepository kezeli a párbeszéd végét is, ha egy párbeszéd lezárult, eltávolítja az összes ahhoz tartozó pillanatképet, és kitakarítja a memóriát, ezzel elkerülve a kulcsok esetleges ütközéseit, vagy ugyanazon kulcshoz tartozó pillanatképek többszöröződését.

6.12. Komponensek

A JSF keretrendszer egyik legnagyobb előnyét a komponensek szolgáltatják, segítségükkel ugyanis nagyon gyorsan, wysiwyg módon fejleszthetőek webalkalmazások. Az 1.2.-ben még csak két tag könyvtár volt, a mag és a html, ezekbe volt szétosztva az összes komponens, továbbá lehetett használni különböző fejlesztők által létrehozott kiegészítő komponenseket is, amik legtöbbször valamilyen komolyabb eszközrendszerrel bővítették a JSF szegényes funkcionalitással rendelkező komponenseit. Ilyen kiegészítő komponensekből áll a Spring Faces is, ugyanis a SWF maga nem tartalmaz semmilyen komponenst a nézetek összeállítására, csak a navigációs logikával foglalkozik. A Spring Faces komponenskönyvtár a facelet technológiára épül, ami többek között összetett nézetek megvalósítására is használható. Az SF komponensei főleg a kliens oldali validációra és AJAX vezérlésre épültek.

A JSF 2.0.-ba jelentősen újítottak a komponensek területén, főleg a sajt komponensek készítésén. Az 1.2.-ben ha írni akartunk egy komponenst, öt lépést kellett elvégezni: először megírni a komponens tld (tag library definition) fájlját, majd a hozzá tartozó tag beolvasó osztályt, ezt konfigurálni a faces-config.xml-ben, majd megírni a komponens saját kezelő osztályát, és végül regisztrálni az oldalon a névterek között. Mint látható ez elég kellemetlen, és nem túl egyszerű folyamat. A JSF 2.0. viszont változtat ezen, az új, saját komponensek létrehozására szolgáló eszköztárával (custom components). Ez az eszköztár lehetővé teszi saját komponensek készítését egyetlen fájl készítésével, kódolás nélkül.

Nézzünk egy példát a 2.0.-ban saját komponens létrehozásához. Készítsünk egy egyszerű feliratot, ami köszön a hello attribútumában megadott szöveggel. Ehhez csak egy xhtml fájlt kell létrehozni a resource kezelés szerinti könyvtárban. Legyen ez most a resources/helloComponent/hello.xhtml. Ezen belül deklarálni kell a `http://java.sun.com/jsf/composite` névteret, majd az ebben található `<composite:interface>` elem segítségével a komponensünk metaadatait definiálhatjuk (attribútumok, facet-ek, eseményfigyelők), a `<composite:implementation>` elemben pedig a komponenst összeállító kódot. (6.39. kódrészlet)

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:composite="http://java.sun.com/jsf/composite">
  <body>
    <composite:interface>
      <composite:attribute name="hello" />
    </composite:interface>
    <composite:implementation>
      <h:outputText value="Hello, #{cc.attrs.hello}!" />
    </composite:implementation>
  </body>
</html>
```

6.39. kódrészlet

Az így elkészült komponensünket utána tetszőleges nézeten felhasználhatjuk, a következő szabályok szerint:

- A komponenst tartalmazó xhtml oldal neve lesz a komponens neve.
- A komponenst tartalmazó könyvtár nevét a `http://java.sun.com/jsf/composite/` cím után kell illeszteni

Az előző példához visszatérve a komponenst használni a 6.40. kódrészlet szerint lehet.

A JSF 2.0.-ban további újdonság hogy a facelet technológiát belevették a szabványba, így azt használhatjuk az oldalak felépítéséhez. Címszavakban a használatáról: Az `ui:composition` való részoldalak definiálásához, ezeket az oldalakat illeszthetjük majd aztán be az `ui:insert` helyekre, a megfelelő nevek segítségével, amely neveket az `ui:define` elemmel adhatunk meg az `ui:composition` elemen belül. Az `ui:composition template` attribútumában adhatjuk meg, melyik mintaoldalba szeretnénk beilleszteni az elemeket. A részegységeket csak simán csatolni is lehet az oldalba az `ui:include` segítségével, ezeket szokták snippeteknek nevezni. (az itt elmondottak egy része felismerhető a 6.40. kódrészleten is).

```
 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:helloComp="http://java.sun.com/jsf/composite/helloComp"
    template="/templates/template.xhtml">
  <ui:define name="title">
      Hello
  </ui:define>
  <ui:define name="content">
    <h:form id="myform">
      <helloComp:hello hello="Tibi"></helloComp:hello>
    </h:form>
  </ui:define>
</ui:composition>
```

6.40. kódrészlet

6.13. Események

A JSF 1.2.-ben két fő esemény volt. A FacesEvent a komponensek által kiváltható eseményeket foglalta magába, mint a komponens szerkesztésekor kiváltódott vagy a gombok által adott események. Míg a PhaseEvent események az életciklus lépései között, a fázis elején és végén, váltódtak ki. Ezekre az eseményekre a PhaseListener interfész implementálásával és az implementáló osztály faces-config.xml-ben történő konfigurálásával lehetett feliratkozni.

A JSF 2.0. kibővíti ezt a két fajta eseményt egy harmadikkal, a SystemEvent-el. Ezt tetszőleges objektum kiválthatja, nem alkalmazás specifikus esemény. A rendszereseményeknek két kategóriája van: a globális rendszeresemények és a komponens

rendszeresemények. A globális rendszeresemények az alkalmazásban bekövetkező különböző eseményekről szolgáltatnak információkat, mint például az alkalmazás inicializálása vagy megsemmisítése, feliratkozni rájuk az `Application.subscribeToEvent()` metódussal lehet. A komponens rendszeresemény specifikus az egyes komponensek esetén, például ilyen esemény következik be a komponens nézethez adásakor, renderelésekor vagy validálásakor. Ilyen eseményre az `UIComponent.subscribeToEvent()` metódussal lehet feliratkozni. A komponens szintű rendszeresemény felgyűrűzhet egészen az alkalmazás szintjére is, így lehetővé téve például a komponensek eseményeinek monitorozását. A feliratkozáshoz a SystemEventListener interfészt kell implementálnunk, és regisztrálnunk az előbb említett két metódus segítségével. Végül rendszereseményt kiváltani kódból a `Application.publishEvent()` segítségével lehet.

A JSF 2.0. a komponens szintű rendszeresményekre való regisztrálás megkönnyítése érdekében bevezette az `<f:event>` elemet. Segítségével a nézeten deklaratívan megadhatjuk a regisztrálni kívánt eseményfigyelőt a komponensen belül.

A SWF ezzel szemben az eseményeket egyetlen figyelő interfésszel oldja meg, és az összes esemény esetén a figyelő valamelyik metódusa fog lefutni. Ez az egyetlen interfész a FlowExecutionListener, amivel a flow végrehajtási állapotáról kaphatunk információt. Az események lehetnek például az állapot indulása, felhasználói esemény bekövetkezése, nézet renderelése satöbbi. Figyelőt regisztrálni a flow beállításainál lehet a `flow-executor` elemen belül a `flow-execution-listeners` segítségével (6.41. kódrészlet).

```
<flow:flow-executor id="flowExecutor">
  <flow:flow-execution-listeners>
    <flow:listener ref="jpaFlowExecutionListener"/>
  </flow:flow-execution-listeners>
</flow:flow-executor>
```

6.41. kódrészlet

Mivel a FlowExecutionListener túl sok metódust tartalmaz, mindet implementálni kényelmetlen lenne, és legtöbbször szükségtelen is. Ezért az SWF biztosít egy absztrakt osztályt, ami implementálja üresen az összes metódust. Ez az osztály az FlowExecutionListenerAdapter, és ezt kiterjesztve csak a ténylegesen szükséges metódusokat kell implementálni.

6.14. Kivételkezelés

A JSF 2.0. egy új eszközt biztosít az életciklus folyamata alatt keletkezett váratlan kivételek együttes kezelésére, ez ugyanis nem volt benne az 1.2.-ben, ott, ha bekövetkezett egy kivétel, nehéz volt kideríteni mi a kiváltó ok. Ez az új eszköz az ExceptionHandler, és az életciklus lefutása utáni globális kivételkezelést tesz lehetővé, viszont nem tartalmazza az alkalmazás indulásakor és végekor bekövetkező kivételeket. Az alapértelmezett ExceptionHandler lényegében kicsomagolja az előfordult kivételt a sorból, majd, mint ServletExceptiont újra eldobja, így lehetővé téve a web.xml konfigurációs fájlban az

`<error-page>` direktíva használatát. Ezen viselkedés előtt az alkalmazásnak van lehetősége lekezelni a hibákat az ExceptionQueuedEvent rendszeresemény figyelésével. Az ExceptionQueuedEvent egy csomagoló osztály, a keletkezett kivétel és környezete becsomagolására.

Az SWF is biztosít központi kivételkezelést, a FlowHandler segítségével. Ezt az interfészt kiterjesztő osztály példányát, mint már a konfigurációs részben írtam, minden egyes flowhoz hozzárendelhetjük. Az interfész által definiált `handleException()` metódus tetszőleges implementálásának segítségével lekezelhetjük a flowban bekövetkezett, le nem kezelt kivételeket.

Az SWF viszont biztosít egy más fajta kivételkezelést is, méghozzá az állapotátmenetekkor. Ha az állapotban valamilyen kivétel következik be, akkor a `transition` elem `on-exception` attribútumában megadva a kivétel teljes elérési útját és nevét, a normális működésnek megfelelően zajlik tovább a flow végrehajtása. (6.42. kódrészlet)

```
<view-state id="elso" view="elso.jspx">
  <transition on-exception="package.ArbitraryException" to="viewName"/>
</view-state>
```

6.42. kódrészlet

További kivételkezelésre ad lehetőséget a FlowExecutionExceptionHandler. Ezt az interfészt implementáló osztályunkat a flowban vagy az állapotokban regisztrálhatjuk az `exception-handler` tag segítségével. Ezt a hibakezelési módszert viszont sokkal bonyolultabb megvalósítani, ugyanis ilyenkor a teljes irányítás át lesz adva az osztályunknak a flow működésének összes információjával, így ha nem figyelünk könnyű a flowt inkonzisztens állapotban hagyni.

Még egy kivétel kezelési lehetőség van, mégpedig az előző fejezetben leírt FlowExecutionListener regisztrálása. Ennek az interfésznek ugyanis van egy `exceptionThrown()` metódusa, ami kivétel keletkezésekor fut le a regisztrált figyelőkön.

6.15. Annotációk

A JSF 2.0.-ban a konfiguráción sokat egyszerűsítettek. Most már nem kell mindenhez az XML, helyette sok dolgot annotációk segítségével is beállíthatunk (6.43. táblázat), és ezzel a lehetőséggel jelentősen csökkenthetjük a faces-config.xml konfigurációs fájl méretét.

Továbbá használhatunk annotációkat a beanek életciklusára vonatkozóan, mint a `@PostConstruct` és a `@PreDestroy` melyek segítségével inicializálás után és megsemmisítés előtt lefutó metódusokat adhatunk meg. Ezt a két annotációt egyébként a Spring is értelmezi. Annotációval regisztrálhatunk eseményfigyelőt is, ha az osztály, amin használjuk, implementálja a SystemEventListener interfészt. Komponenseken és renderelő

osztályokon használhatjuk a `@ResourceDependency` annotációt, amivel a nézethez szükséges resourecokat lehet megadni, mint azt a „Resource kezelés” fejezetben már írtam.

JSF beállítás	Annotáció
`<managed-bean>`	`@ManagedBean`
`<managed-bean-name>`	- name
`<managed-bean-scope>` - application - session - custom - view - request - none	 @ApplicationScoped @SessionScoped @CustomScoped @ViewScoped @RequestScoped @NoneScoped
`<managed-property>`	`@ManagedProperty`
`<property-name>`	- name
`<value>`	- value
`<referenced-bean>`	`@ReferencedBean`
`<referenced-bean-name>`	- name
`<component>`	`@FacesComponent`
`<component-type>`	- value
`<converter>`	`@FacesConverter`
`<converter-id>`	- value
`<converter-for-class>`	- forClass
`<render-kit>`	
`<renderer>`	`@FacesRenderer`
`<renderer-type>`	- rendererType
`<component-family>`	- componentFamily
`<render-kit-id>`	- renderKitId
`<validator>`	`@FacesValidator`
`<validator-id>`	- value
`<behavior>`	`@FacesBehavior`
`<behavior-id>`	- value

6.43. táblázat

A JSF 2.0. továbbá támogatja a Java EE 5 keretrendszerben bevezetett annotációk használatát, amik a következők: `@Resource`, `@Resources`, `@EJB`, `@EJBs`, `@WebServiceRef`, `@WebServiceRefs`, `@PersistenceContext`, `@PersistenceContexts`, `@PersistenceUnit`, `@PersistenceUnits`. Valamint a JSR-303-ban leírt annotációkkal való validálást is.

Ezzel szemben a SWF-ben nem lehet annotációkkal definiálni semmit, ami azt eredményezi, hogy a flow definíciós fájlok méretét nem lehet ilyen módon csökkenteni. Az annotációk hiánya csak a Spring Web Flowra értendő, a Spring magban és az MVC-ben is a 3.0. verziótól szinte mindent lehet annotációkkal is definiálni, mint azt már korábbi fejezetekben írtam. A Spring mag is ismeri a JSR-303-as validációs, és a JSR-330 injektálásra

vonatkozó szabványt valamint a Java EE 5 annotációk egy részét (JSR-299), pontosabban a JPA által definiált `@PersistenceContext` és `@PersistenceUnit` annotációkat.

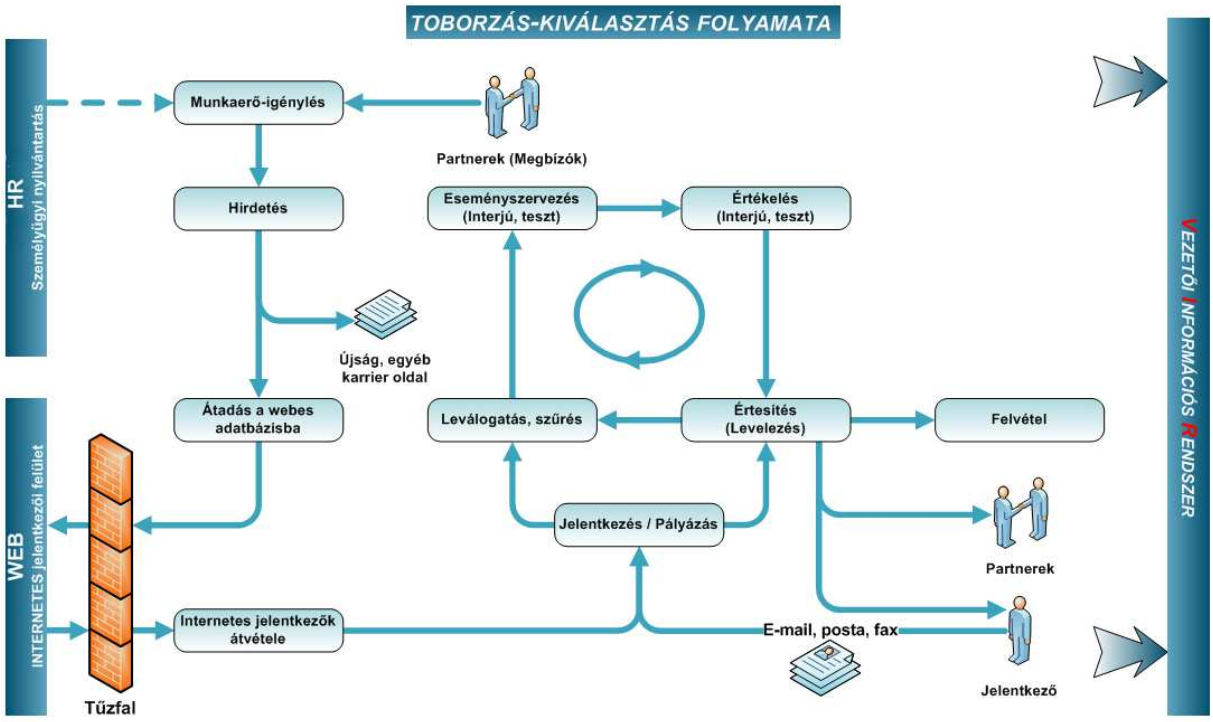
A Springben annyi mindent lehet annotációval definiálni, több féle képen is, hogy ha itt elkezdeném kifejteni azokat, több oldalt megtöltene. Ehelyett inkább leírok néhányat a nagyon alapvetőek közül. A beaneket definiálni a `@Component` vagy annak egy specializált (`@Controller`, `@Repository`, `@Service`) változatával lehet. A beanek függőségeit az `@Autowired` annotációval lehet, ami elég sokoldalúan használható, az injektálás mindig típus szerint fog eldőlni, és injektálhatjuk az összes ugyanolyan típusú beant is kollekcióként. Szükséges függőséget a `@Required` segítségével adhatunk meg. Az `@Autowired` helyett lehet használni a `@Resource` JSR-250-es annotációt, amivel név szerint injektálhatunk. A scopeot megadni `@Scope` segítségével lehet, és a BeanDefinition interfészben megadott scope konstansokkal. Az adatkezelési rétegnél is további jelentős mennyiségű annotációt találhatunk, ahol az egyik leglényegesebb a `@Transactional`, ami tranzakcióba foglalja a metódust, amin alkalmaztuk. Az MVC kontrollereinek definiálása is közel teljesen megtehető annotált formában, de ezek nagy részét már az azt leíró fejezetben kifejtettem.

7. A webalkalmazás

Sokat gondolkoztam azon, mit valósítson meg a programom, végül a gyakorlatomból ihletet merítve egy egyszerűbb HR toborzási modul egy részét választottam ki.

A gazdasági szervezetek munkaerő állománya folyamatos változásban van mind annak számát és annak minőségét tekintve. Ilyen változások sok okból keletkezhetnek, mint például a gazdasági szervezet tudatos döntései alapján, vagy szervezeten kívül álló hatások következményeiként, mint politikai, környezeti vagy gazdasági hatások. Így a szervezet nagy valószínűséggel szemben fogja magát találni azzal a problémával, hogy az általa alkalmazott munkaerő összetételét a változó környezethez igazítsa, ezért egy állandóan változó létszámszerkezettel kell számolnia.

A létszám változása miatt, a szervezetnek tisztában kell lennie azzal, hogy milyen szintű és mennyiségű tudás szükségeltetik a napi tevékenységek elvégzéséhez, és céljai eléréséhez. Mivel a célok gyakran változnak, az ezek eléréshez megszerzendő többlettudás és kompetencia idő és ráfordítás-igényes, aminek csökkentése a munkaerő-állomány mellé vagy helyett felvett, a munkaerőpiacon fellelhető magasabb tudás és kompetenciaszintű munkavállalókkal lehetséges. Ezen tevékenységet a toborzás és kiválasztás folyamatának nevezik (7.1. ábra). Ez a tevékenység jellemzően jelentős adminisztratív terhet ró annak végzőjére, és magas fokú szervezőkészséget igényel.



7.1. ábra. Toborzás folyamata.

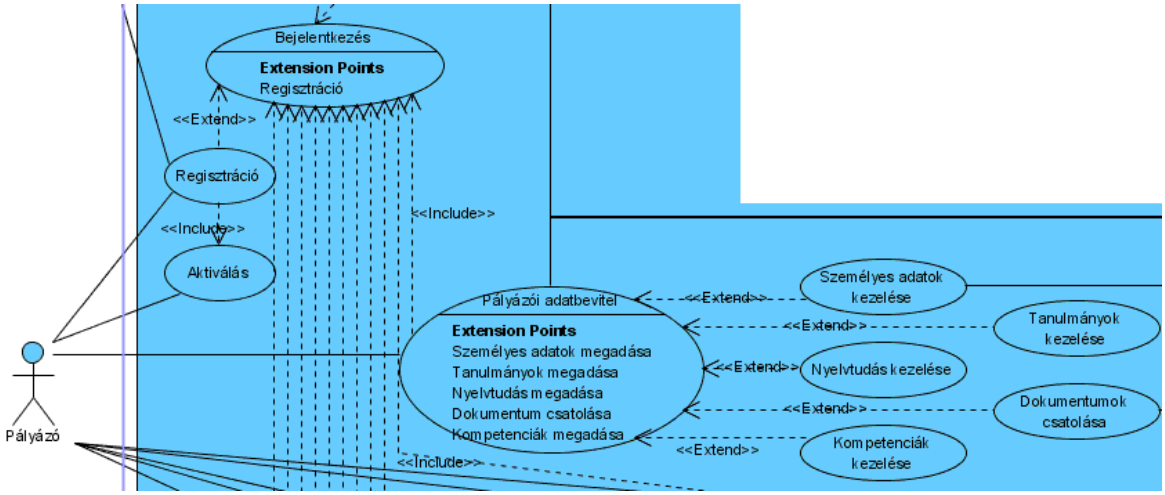
Egy ilyen rendszerrel kapcsolatban felmerülő fő követelmények a következők:

- Munkaerő igény keletkezése
  - Kiválasztási folyamat indítása
- Munkaerő igény jóváhagyása/elutasítása
- Pályázat kiírás:
  - A hirdetés összeállítása
  - Hirdetés különböző médiákban, adatbázisokban
- Pályázók projekthez rendelése
  - Jelentkezői adatbázis
  - Beérkezett anyagok összehasonlítása az igényekkel
- Interjúk szervezése, interjúsablonok, tesztek, résztvevők, eredmények:
  - Interjúk
  - Tesztek
- Felvétel/elutasítás
- Munkaügyi adatok kezelése

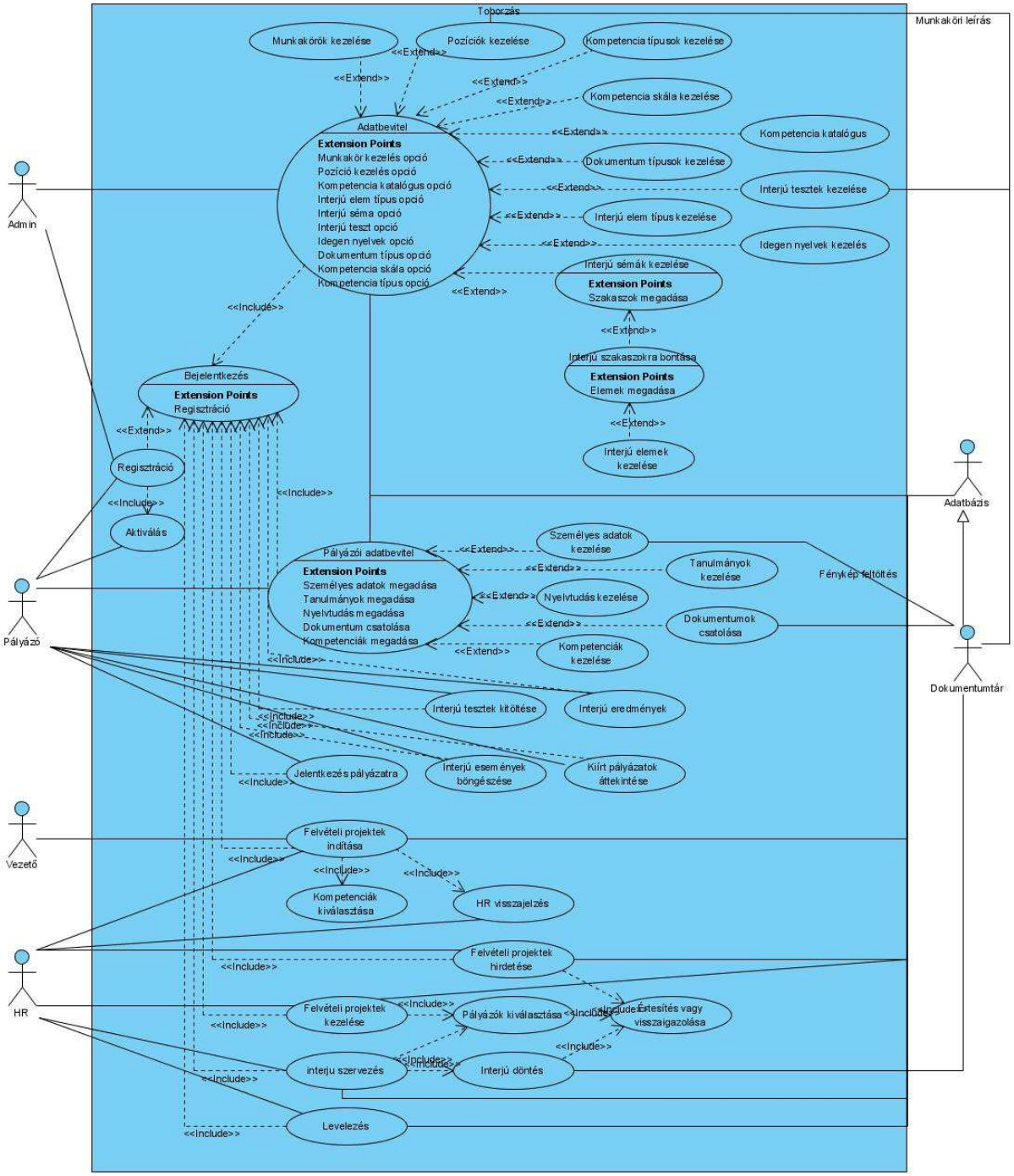
Ebből a nagyobb követelményrendszerből a jelentkezői adatbázis kialakítását készítettem el a szakdolgozat keretén belül. Ennek a résznek a következő részkövetelményeknek kell megfelelnie:

- A pályázónak regisztrálnia kell tudnia az adatbázisba
- A pályázónak be kell tudnia jelentkezni.
- A pályázónak meg kell tudni adni adatait.
  - Személyes adatait
  - Végzettségét
  - Korábbi munkahelyeit

A toborzás modul use case diagramja a 7.2. ábrán látható teljes egészben. Ebből csak a 7.3. ábrán látható részlet lényeges a szakdolgozat szempontjából.



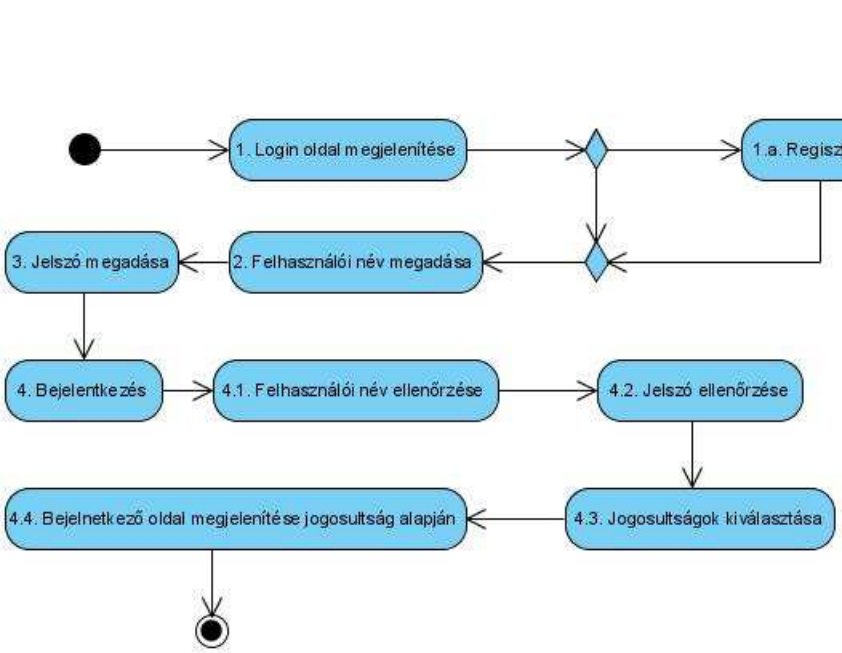
7.3. ábra. Az elkészült páylázói adatbázis use case diagram részlete.



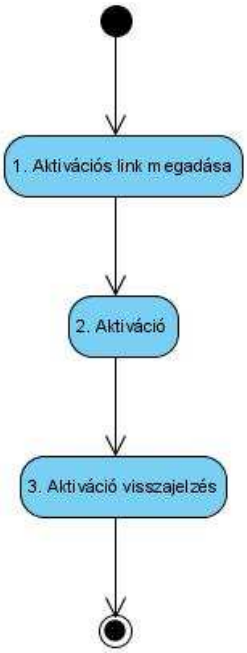
7.2. ábra. A rendszer teljes use case diagramja

A pályázó regisztrációja, vagy belépése a következő módon zajlódhat le: Ha van felhasználó neve, akkor a nyitó oldalon bejelentkezhet, és az adatainak böngészésével folytathatja tevékenységét. Viszont ha nincs még felhasználó neve, regisztrálnia kell magát, amit a regisztráció gomb megnyomása után az adatainak kitöltésével megtehet. Itt minimális validáció történik, már a kliens oldalon is. Miután megadta a megfelelő adatokat egy aktivációs link kerül elküldésre e-mailen keresztül (a szakdolgozat mellé leadott programban csak kiírja a linket egy felugró ablakban). Az aktiváció után a pályázó beléphet az adatainak szerkesztéséhez. A folyamat activity diagramjai a 7.4., 7.5., 7.6. ábrán láthatóak. Érdemes

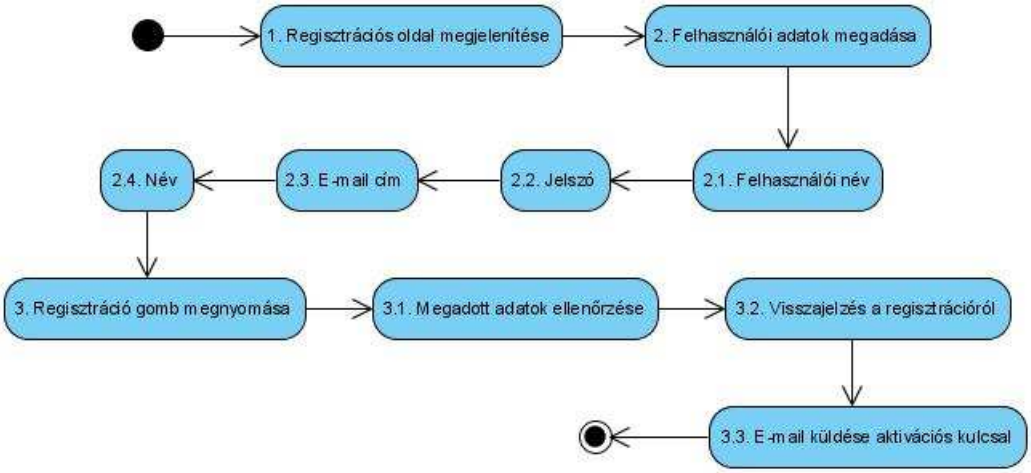
megjegyezni, hogy a bejelentkezési logika a Spring Security segítségével készült, amit a login-flow indít el. Míg az aktiváció MVC és JSF 1.2 együttes használatával történik.



7.4. ábra. A bejelentkezés folyamata

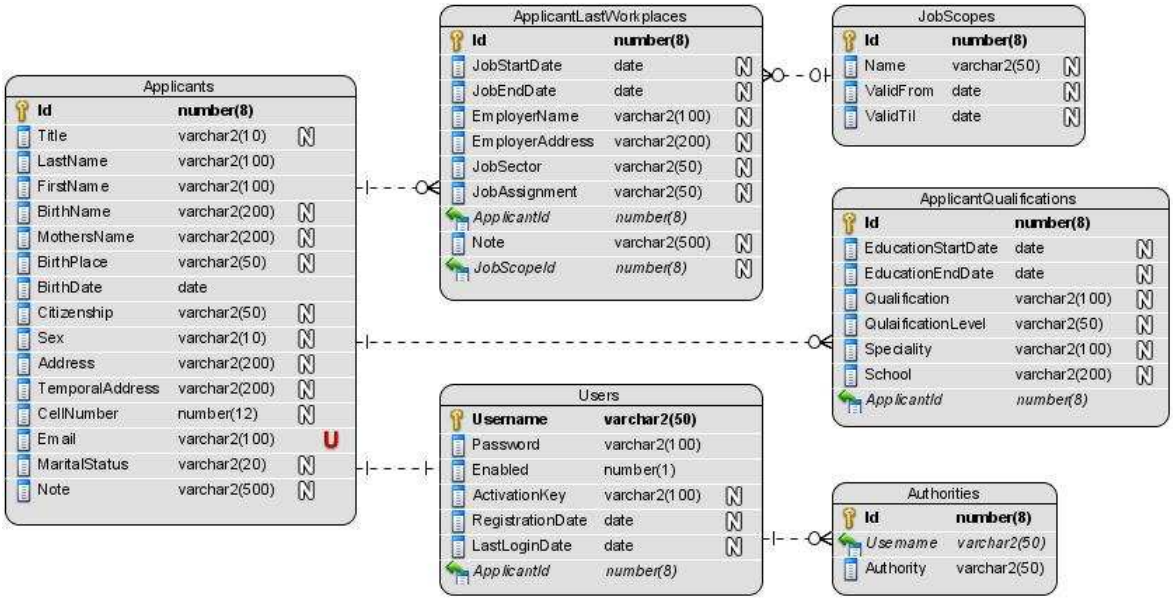


7.5. ábra. Aktiváció

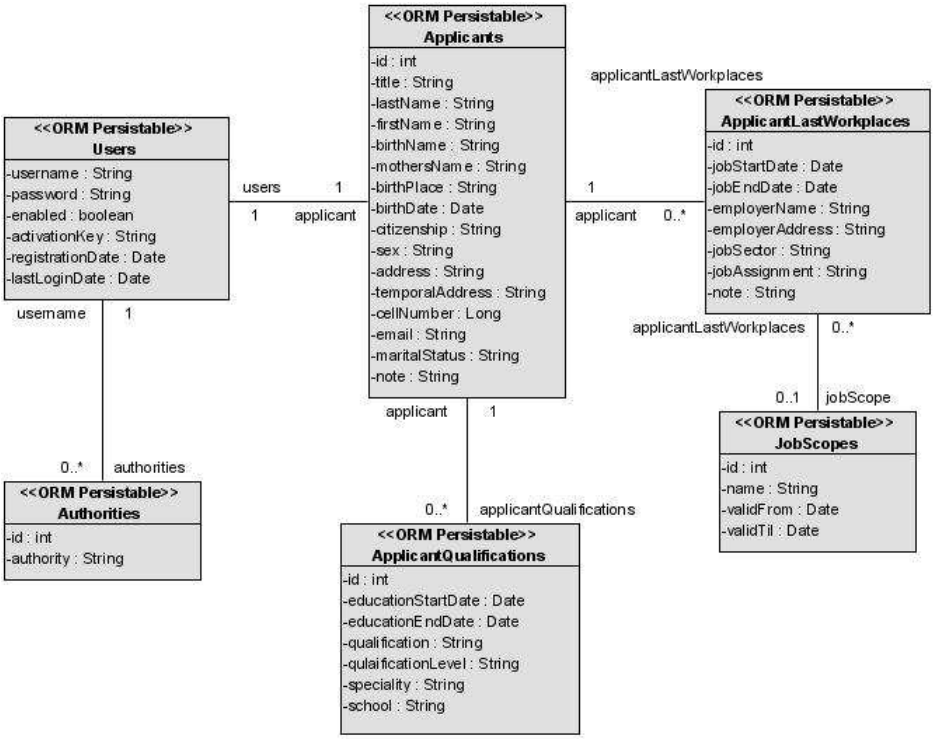


7.6. ábra. A regisztráció folyamata

A use case diagram segítségével készült, a pályázó adatainak tárolásához szükséges adatbázistervek a 7.7. ábrán (ER diagram) és a 7.8. ábrán (ORM diagram) láthatóak.



7.7. ábra. Az elkészített pályázói adatbázis ER diagramja



7.8. ábra. Az elkészített pályázói adatbázis ORM diagramja

Végül az elkészült webalkalmazásból következzen két kép, a login oldalé (7.9. ábra) és a pályázó adatainak megjelenítéséért felelős oldalé (7.10. ábra). Az alkalmazás eclipse segítségével készült, mint egyszerű dinamikus web projekt. Az adatbázis memória alapú HSQLDB, JPA szolgáltató a Hybernate, a biztonságért a Spring Security felel, a tranzakciókat a Spring kezeli lokális deklaratív módszerrel, annotációs deklarációval. A nézet megjelenítéséért a JSF 1.2, míg a navigációért és az üzleti logika egy részéért a Spring Web Flow felelős. Az alkalmazásban megtalálható kliens oldali és szerver oldali validáció is. A

kliens oldali JavaScript segítségével fut, ha ez a kliensnél ki van kapcsolva, akkor is működik az alkalmazás. A felugró ablakok AJAX vezéreltek. Az oldalakon használt feliratok mind resource fájlból töltődnek be, de csak a magyar nyelv támogatott.

7.9. ábra. A bejelentkező oldal

7.10. ábra. A pályázó adatainak megjelenítése

8. Összefoglalás

Úgy érzem az SWF és a JSF 2.0 összehasonlítása megfelelően kimerítőre sikerült, a dolgozat a célját elérte, de lehetne még bővíteni. A Spring és JSF alapjaira úgy gondolom megfelelő hangsúlyt fektettem, és sikerült érthetően leírnom pár oldal alatt azok lényegét, és az azok által biztosított lehetőségeket is. Ugyan az SWF és JSF 2.0 technológiáról az itt leírtakon túl is bőven lehetne mit írni, ugyanis vannak részek, amik kimaradtak, főleg a technológiaspecifikus dolgok, de ezekre már nem jutott idő. A leírtakon felül mindkét keretrendszer nyújt néhány érdekes lehetőséget, amiket talán egy másik szakdolgozatban lehetne tovább fejtegetni.

Mint már a bevezetőben is írtam, nem vonok le mély következtetéseket a korábban leírtakból, ezt mindenkinek magának érdemes megtennie. Egy dolgot viszont hozzáfűznék, mégpedig azt, hogy a JSF 1.2-nek rengeteg hiányossága volt, amit az SWF ki is javított, azaz a JSF 1.2-vel érdemes az SWF-et kombinálni, vannak ugyan apró hibák, de szerintem jelentős többletet nyújt ezek együttes használata. Viszont a JSF 2.0 megjelenésével, úgy gondolom mindkét keretrendszer nagyjából egy szintre került, mint az látható is volt a leírásokban. Szinte minden egyes szempont megtalálható volt mindkét technológiában, csak legfeljebb máshogy, más kód segítségével lehetett megvalósítani azokat.

9. Irodalomjegyzék

[1] David Geary, Cay Horstmann. *Core JavaServer™ Faces Second Edition.* Prentice Hall, 2007.

[2] Jonas Jacobi, John R. Fallows. *Pro JSF and Ajax: Building Rich Internet Components.* Apress, 2006

[3] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Alef Arendsen, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, Arjen Poutsma, Chris Beams, Tareq Abedrabbo, Andy Clement. *Spring Framework Reference Documentation 3.0.* Spring Source, 2004-2009. http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/

[4] Spring Source. *Spring Framework 3.0.x API.* http://static.springsource.org/spring/docs/3.0.x/javadoc-api/

[5] Keith Donald, Erwin Vervaet, Jeremy Grelle, Scott Andrews, Rossen Stoyanchev. *Spring Web Flow Reference Guide, Version 2.0.9.* Spring Web Flow, 2010. http://static.springsource.org/spring-webflow/docs/2.0.x/reference/html/

[6] Spring Source. *Spring Web Flow 2.0.x API.* http://static.springsource.org/spring-webflow/docs/2.0.x/javadoc-api/

[7] Jan Machacek, Aleksa Vukotic, Anirvan Chakraborty, Jessica Ditt. *Pro Spring 2.5.* Apress, 2008.

[8] Seth Ladd, Darren Davison, Steven Devijver, Colin Yates. *Expert Spring MVC and Web Flow.* Apress, 2006.

[9] Sven Lüppken, Markus Stäuble. *Spring Web Flow 2 Web Development.* Packt Publishing, 2009.

[10] Erwin Vervaet. *The Definitive Guide to Spring Web Flow.* Apress, 2008.

[11] Ed Burns, Roger Kitain, editors. *JavaServer™ Faces Specification Version 2.0 Final Draft Candidate 20090506.* https://javaserverfaces-spec-public.dev.java.net

[12] Sun Microsystems, Inc. *JavaServer Faces API (2.0).* https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/index.html

[13] Andy Schwartz. *Whats New in JSF 2.0.* http://andyschwartz.wordpress.com/2009/07/31/whats-new-in-jsf-2/